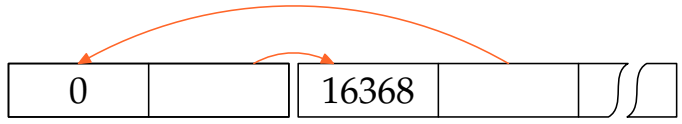
- Im folgenden wird eine sehr einfache Speicherverwaltung vorgestellt, die
  - das Belegen und Freigeben von Speicher unterstützt,
  - freigegebene Speicherflächen wieder zur Verfügung stellen kann und auch
  - in der Lage ist, mehrere hintereinander freigegebene Speicherflächen zu einer größeren Freifläche zusammenzufügen, um Fragmentierung zu vermeiden.
- Der vorgestellte Algorithmus ist in der Literatur bekannt als *circular first fit*. Eine ähnliche Fassung wurde bereits im ersten C-Buch von Kernighan und Ritchie vorgestellt.

alloc.c

```
typedef struct memnode {
   size_t size;
   struct memnode* next;
} memnode;
```

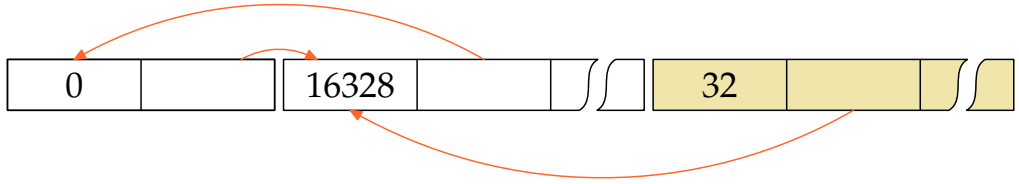
- Allen freien oder belegten Speicherflächen geht ein Verwaltungsobjekt des Typs *memnode* voraus.
- *size* gibt die Größe der Speicherfläche an, die diesem Verwaltungsobjekt unmittelbar folgt, jedoch ohne Einberechnung des Speicherbedarfs für das Verwaltungsobjekt
- *next* verweist
  - bei freien Speicherflächen auf das nächste Verwaltungsobjekt im Ring freier Speicherflächen
  - bei belegten Speicherflächen zum unmittelbar vorangehenden Speicherelement, egal ob dieses frei oder belegt ist.

Ring der freien Speicherflächen



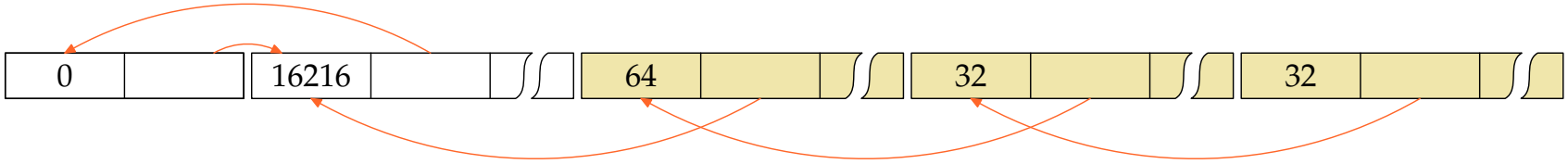
- Alle freien Speicherflächen sind in einem Ring organisiert.
- Ein Ring ist bei dem gewählten Algorithmus *circular first fit* notwendig, weil nicht immer von Anfang an gesucht wird, sondern dort die Suche begonnen wird, wo sie zuletzt endete.
- Damit sich der Ring nicht auflöst, wenn alle zur Verfügung stehenden Speicherflächen vergeben sind, gibt es ein spezielles Ring-Element, das nur aus einem Verwaltungsobjekt besteht, aber keinen eigentlichen Speicherplatz anbietet.
- Das Diagramm zeigt zwei Ringelemente. Links ist das spezielle Element (mit $size = 0$) und rechts ein Element, das noch 16368 Bytes frei hat.

Belegung der ersten Speicherfläche

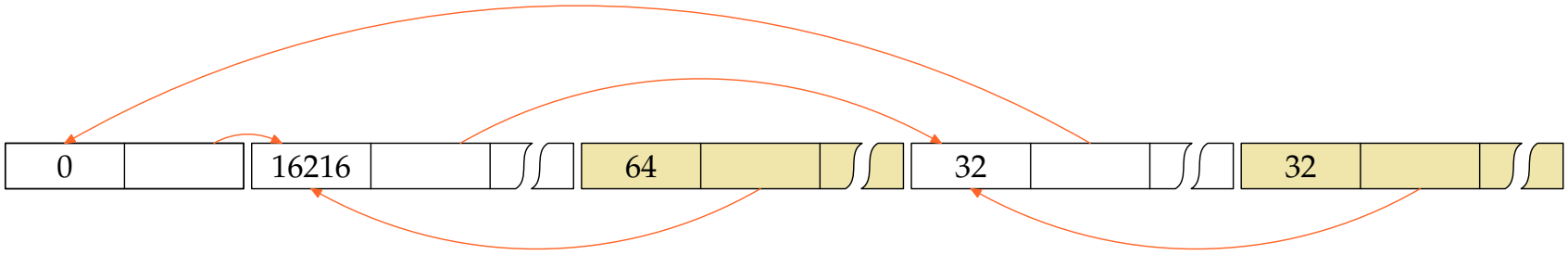


- Wenn nun 32 Bytes belegt werden sollen, wird nach einem Element im Ring der freien Speicherflächen gesucht, das mindestens 32 Bytes anbietet. In diesem Beispiel gab es nur das ganz große.
- Da noch Speicher übrigbleibt, wird das Element geteilt: Das Ende wird für die zu vergebende Speicherfläche Platz reserviert mitsamt einem Verwaltungsobjekt und bei dem entsprechenden freien Element wird *size* verkleinert, von 16368 auf 16328 (unter der Annahme, dass ein Verwaltungsobjekt 8 Bytes belegt und somit $32 + 8 = 40$ Bytes benötigt wurden).
- Bei dem Verwaltungsobjekt für die belegte Speicherfläche verweist der *next*-Zeiger auf das im Speicher unmittelbar davorliegende Element; das ist hier noch das Element aus dem Ring der freien Speicherflächen.

Belegung weiterer Speicherflächen

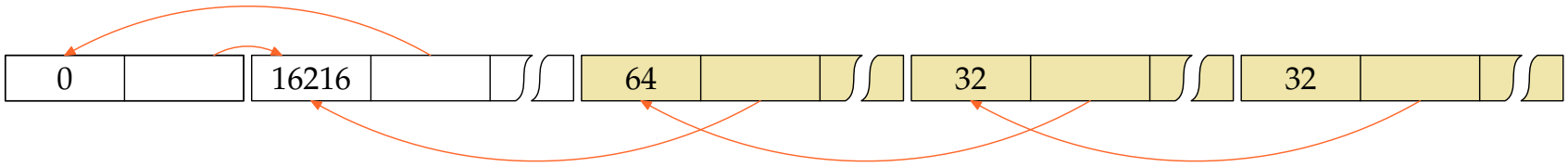


- Inzwischen wurden zwei weitere Speicherflächen belegt, zuerst noch einmal mit 32, dann mit 64 Bytes.
- Diese wurden allesamt dem großen Ringelement der freien Speicherflächen entnommen.
- Zu beachten ist, dass die Verwaltungsobjekte der belegten Speicherflächen jeweils auf das im Speicher unmittelbar davorliegende Element verweisen, egal ob dies frei ist oder nicht. Auf diese Weise ist es bei einer Freigabe recht einfach, ein freigegebenes Element mit einem bereits freien Element in der unmittelbaren Nachbarschaft zu vereinigen.



- Wenn eine belegte Speicherfläche freigegeben wird, wird ausgehend von dem freiwerdenden Element solange die Kette der davorliegenden Elemente verfolgt, bis das erste Ring-Element vorgefunden wird, das eine freie Speicherfläche repräsentiert.
- Die freigegebene Speicherfläche wird unmittelbar dahinter eingehängt.
- Bei dem Ring der freien Speicherflächen bleibt so immer die Ordnung entsprechend der Lage im Speicher erhalten. Nur auf diese Weise ist es später möglich, benachbarte freie Flächen wieder zu größeren freien Flächen zusammenzulegen.

Wie wird eine freie Speicherflächer erkannt? 214



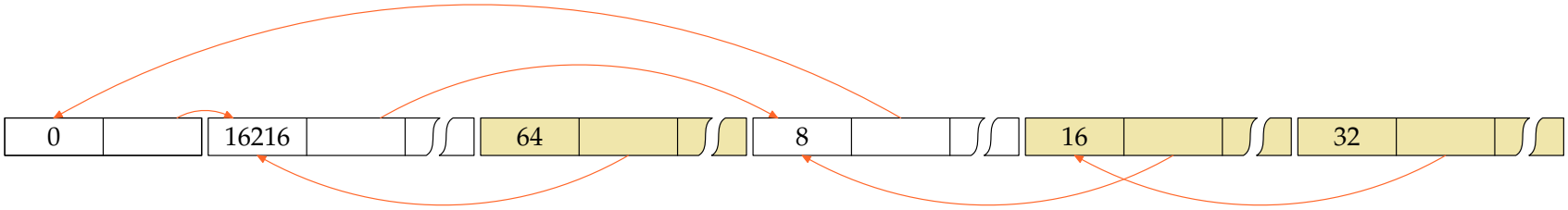
- Wenn bei einer Freigabe das nächste vorangehende freie Element gesucht wird, müssen wir unterscheiden können, ob ein Element frei oder belegt ist.
- Eine Möglichkeit wäre es, etwa bei *size* das niedrigstwertige Bit entsprechend zu setzen. Die Größe muss immer die Alignment-Anforderungen berücksichtigen und entsprechend darf eine Größe nie ungerade sein.
- In diesem einfachen Beispiel mit nur einem großen Speicherblock und einem Spezialelement geht es aber auch ohne diesen Trick...

Wie wird eine freie Speicherflächer erkannt? 215

alloc.c

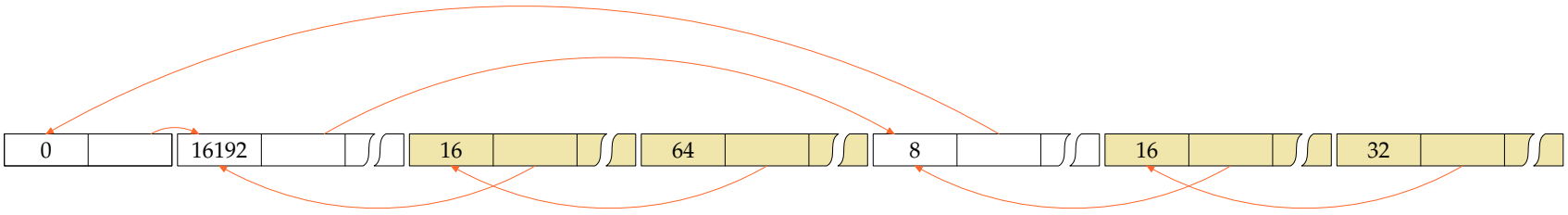
```
bool is_allocated(memnode* ptr) {
   if (ptr == root) return false;
   if (ptr->next > ptr) return false;
   if (ptr->next != root) return true;
   if (root->next > ptr) return true;
   return root->next == root;
}
```

- Das Spezialelement (nennt sich hier *root*) ist immer frei.
- Wenn der Zeiger nach vorne (d.h. zu einer größeren Adresse) weist, dann ist das Element frei.
- Wenn das alles nicht zutrifft und der Zeiger nicht auf das Spezialelement *root* verweist, ist es belegt.
- Wenn *next* auf das Spezialelement *root* verweist, gibt es zwei Fälle:
  - Es ist belegt und das Element liegt unmittelbar hinter dem Spezialelement (am Anfang des großen Blocks) oder
  - es handelt sich um das freie Element mit der höchsten Adresse.



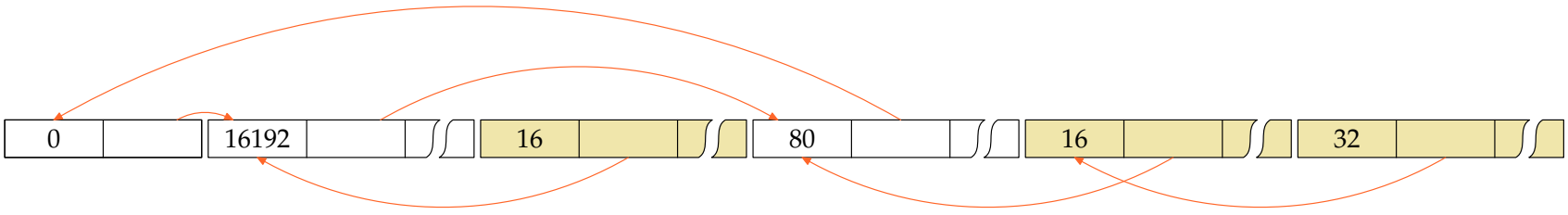
- Wenn nach freien Elementen immer beginnend von dem Spezialelement (*root*) aus gesucht wird, tendiert die Liste der freien Elemente dazu, zu Beginn nur ganz winzige Reste anzubieten, so dass die größeren freien Elemente erst ganz hinten zu finden sind.
- Deswegen wird beim *circular first fit*-Algorithmus die Suche dort fortgesetzt, wo wir zuletzt waren. Und entsprechend dem *first fit* wird die erste Speicherfläche akzeptiert, die genügend Speicherplatz anbietet.
- Das Diagramm zeigt die Situation, wenn 16 Bytes angefordert wurden. Das führte zur Aufspaltung des zuletzt frei gewordenen Elements mit 32 Bytes.

Das Wandern im Ring



- Und wenn erneut Speicher belegt wird, dann beginnt die Suche beim nächsten Element.
- Das ist hier das ganz große freie Element links, von dem wieder etwas am Ende weggenommen wurde.

Das Zusammenlegen freier Speicherflächen



- Wenn ein Element freigegeben wird, dann kann davor und danach jeweils ein freies Element vorliegen, mit dem das neue Element zusammengelegt werden kann.
- In diesem Beispiel fand sich danach ein freies Element.
- Prinzipiell können bei einer Freigabe bis zu drei Elemente zusammengelegt werden.

alloc.c

```
memnode dynmem[MEM_SIZE] = {
   /* bleibt immer im Ring der freien Speicherflaechen */
   {0, &dynmem[1]},
   /* enthaelt zu Beginn den gesamten freien Speicher */
   {sizeof dynmem - 2*sizeof(memnode), dynmem}
};
memnode* node = dynmem;
memnode* root = dynmem;
```

- In dem einfachen Beispiel wird nur Speicher aus dem großen Array *dynmem* vergeben.
- In diesem liegt gleich zu Beginn das Spezialelement, gefolgt von dem großen Element, dem der restliche freie Speicher gehört.
- *root* zeigt immer auf das Spezialelement.
- *node* ist der im Ring herumwandernde Zeiger, der immer auf ein freies Element im Ring verweist.
- Bei einer ernsthaften Implementierung (siehe Übungen und Wettbewerb!) sind dann sukzessive Kacheln vom Betriebssystem zu holen und zu verwalten.

alloc.c

```
memnode* successor(memnode* p) {
   return (memnode*) ((char*)(p+1) + p->size);
}
```

- Das jeweils im Speicher nachfolgende Element zu finden, ist einfach mit Hilfe der Adressarithmetik.
- Zu beachten ist, dass zuerst die Zeigerarithmetik auf Basis des Zeigertyps *memnode*∗ erfolgt mit $p+1$ und dann, um die Größe in Bytes zu addieren, dieser zwischendurch in einen **char**-Zeiger konvertiert werden muss.
- Bei belegten Elementen ist das vorangehende Element immer über den *next*-Zeiger ermittelbar.
- Wenn wir den Ring der freien Elemente durchlaufen, behalten wir immer noch einen Zeiger auf das freie Element davor. Von dem aus können ggf. mit *successor* noch die dazwischenliegenden belegten Speicherelemente durchlaufen werden.
- All diese Tricks stellen sicher, dass die Verwaltungsobjekte nicht zuviel Speicherplatz belegen.

alloc.c

```
void* my_malloc(size_t size) {
   assert(size >= 0);
   if (size == 0) return 0;
   /* runde die gewuenschte Groesse auf
      das naechste Vielfache von ALIGN */
   if (size % ALIGN) {
      size += ALIGN - size % ALIGN;
   }
   /* Suche und Vergabe ... */
}
```

- *malloc* und analog *my_malloc* müssen darauf achten, dass der vergebene Speicher korrekt ausgerichtet ist (Alignment). Am einfachsten ist es hier, alles auf die maximale Alignment-Anforderung auszurichten, das ist hier *ALIGN*.

alloc.c

```
memnode* prev = node; memnode* ptr = prev->next;
do {
   if (ptr->size >= size) break; /* passendes Element gefunden */
   prev = ptr; ptr = ptr->next;
} while (ptr != node); /* bis der Ring durchlaufen ist */
if (ptr->size < size) return 0; /* Speicher ist ausgegangen */
```

- *node* ist der im Ring herumwandernde Zeiger auf ein freies Element.
- Wir setzen *prev* auf *node* und *node* gleich auf das nächste Element. Auf diese Weise kennen wir immer den Vorgänger.
- Danach läuft die Schleife, bis entweder ein passendes freies Element gefunden wurde oder wir den gesamten Ring durchlaufen haben.

alloc.c

```
if (ptr->size < size + 2*sizeof(memnode)) {
   node = ptr->next; /* "circular first fit" */
   /* entferne ptr aus dem Ring der freien Speicherflaechen */
   prev->next = ptr->next;
   /* suche nach der unmittelbar vorangehenden Speicherflaeche;
      zu beachten ist hier, dass zwischen prev und ptr noch
      einige belegte Speicherflaechen liegen koennen
   */
   for (memnode* p = prev; p < ptr; p = successor(p)) {
      prev = p;
   }
   ptr->next = prev;
   return (void*) (ptr+1);
}
```

- Wenn das gefundene freie Element genau passt bzw. zu klein ist, um weiter zerlegt zu werden, muss es aus der Liste der freien Element herausgenommen werden.
- Außerdem muss das korrekte Vorgängerelement gefunden werden, auf das der *next*-Zeiger zu verweisen hat.

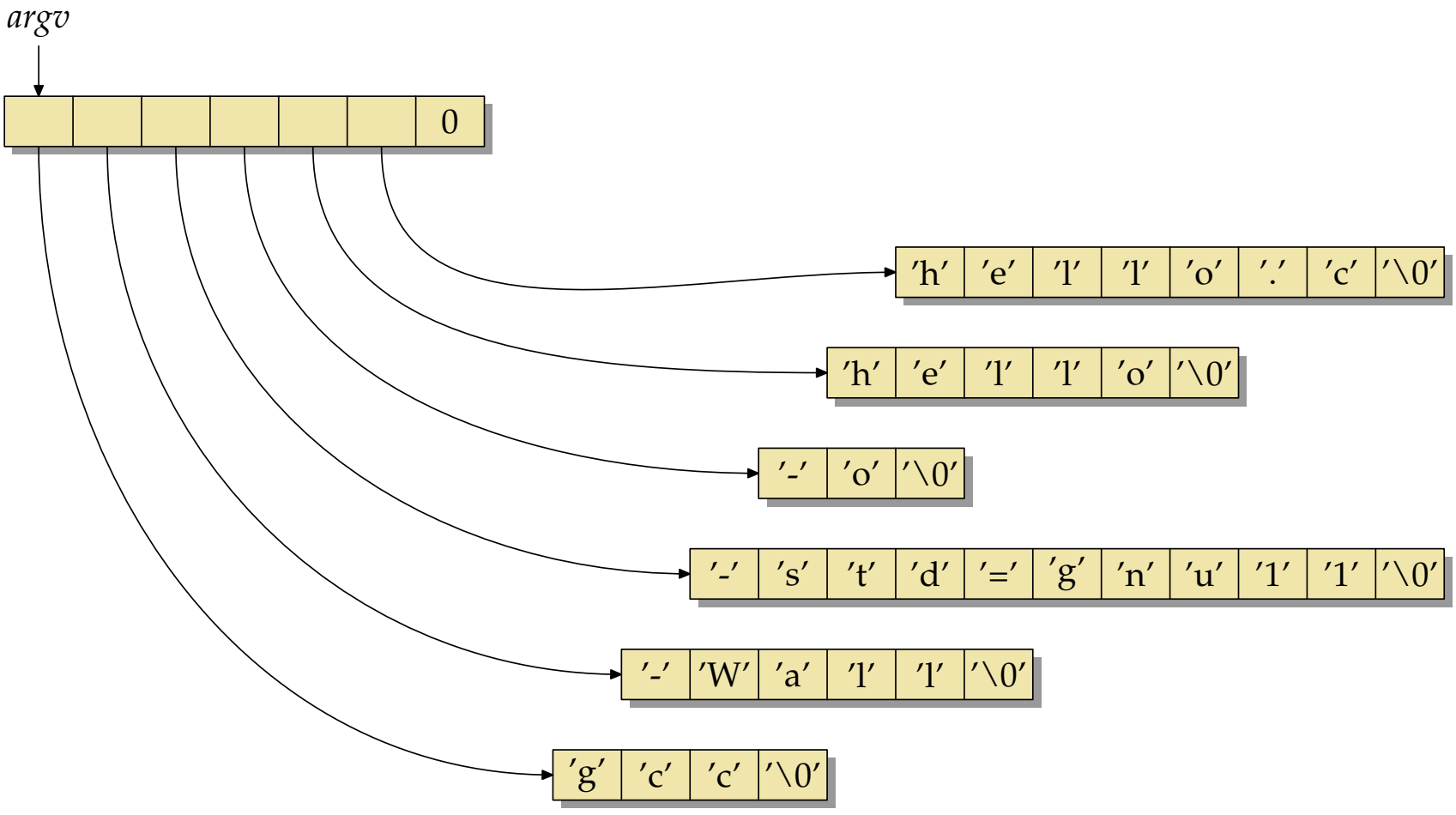
alloc.c

```
   node = ptr; /* "circular first fit" */
   /* lege das neue Element an */
   memnode* newnode = (memnode*)((char*)ptr + ptr->size - size);
   newnode->size = size; newnode->next = ptr;
   /* korrigiere den Zeiger der folgenden Speicherflaeche,
      falls sie belegt sein sollte */
   memnode* next = successor(ptr);
   if (next < dynmem + MEM_SIZE && next->next == ptr) {
      next->next = newnode;
   }
   /* reduziere die Groesse des alten Elements
      aus dem Ring der freien Speicherflaechen */
   ptr->size -= size + sizeof(memnode);
   return (void*) (newnode+1);
```

- Andernfalls ist das gefundene freie Element zu zerlegen in ein weiterhin freies Element am Anfang der Fläche und das neue belegte Element.
- Ferner ist darauf zu achten, dass das folgende Element, falls es belegt ist, auf das neugeschaffene Element davor verweist.

```
int main(int argc, char* argv[]) {
   /* ... */
}
```

- *main* erhält gemäß dem Standard zwei Parameter, *argc* und *argv*, die der Übermittlung der Kommandozeilenparameter dienen.
- *argc* enthält die Zahl der Parameter, wobei der Kommandoname mitgezählt wird.
- *argv* ist ein Zeiger auf ein Array von Zeigern, das auf die einzelnen Kommandozeilenparameter verweist.
- *argv*[0] zeigt auf den Namen des Kommandos, *argv*[1] bis *argv*[*argc*−1] zeigen auf die einzelnen Parameter.
- Die Zeigerliste wird durch einen Nullzeiger terminiert, d.h. *argv*[*argc*] ist 0.



- Dies ist die Repräsentierung der Kommandozeilenparameter für „gcc -Wall -std=gnu11 -o hello hello.c“. *argc* hätte hier den Wert 6.

args.c

```
#include <stdio.h>

int main(int argc, char* argv[]) {
   printf("Command name: %s\n", argv[0]);
   printf("Number of command line arguments: %d\n", argc-1);
   for (int i = 1; i < argc; i++) {
      printf("Argument %d: %s\n", i, argv[i]);
   }
}
```

- Dieses Programm gibt den Kommandonamen (in *argv*[0]) und die übrigen Kommandozeilenparameter aus.
- Der Kommandoname ist normalerweise der Name, mit dem ein Kommando aufgerufen worden ist.

args2.c

```
#include <stdio.h>

int main(int argc, char* argv[]) {
   char* cmdname = *argv++; --argc;
   printf("Command name: %s\n", cmdname);
   printf("Number of command line arguments: %d\n", argc);
   while (argc-- > 0) {
      printf("Argument: %s\n", *argv++);
   }
}
```

- Alternativ können die Kommandozeilenparameter auch sukzessive „konsumiert“ werden, indem entsprechend *argc* gesenkt und *argv* weitergesetzt wird.
- Dann sollte aber die Invariante eingehalten werden, dass *argc* die Zahl der noch unter *argv* verbleibenden Parameter angibt.

mygrep.c

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

int main(int argc, char* argv[]) {
   char line[256];

   if (argc != 2) {
      fprintf(stderr, "Usage: %s pattern\n", argv[0]);
      exit(1);
   }

   while (fgets(line, sizeof line, stdin)) {
      if (strstr(line, argv[1])) {
         fputs(line, stdout);
      }
   }
}
```

- *strstr* sucht nach dem ersten Vorkommen des zweiten Parameter in dem ersten Parameter und liefert, falls gefunden, einen Zeiger darauf zurück, ansonsten 0.

- Per Konvention beginnen Optionen in der Kommandozeile mit dem Minuszeichen „-“.
- Hinter dem Minuszeichen können dann ein oder mehrere Optionen folgen, die typischerweise mit nur einem Buchstaben benannt werden.
- Es gibt auch Kommandos, die längere Optionsnamen unterstützen. Dann können aber Optionen nicht mehr in einem Parameter integriert werden oder die Optionen müssen anders beginnen. GNU-Werkzeuge verwenden dafür gerne das doppelte Minuszeichen „--“.
- Zwei alleinstehende Minuszeichen beenden die Folge der Optionen.
- Danach folgen typischerweise Pflichtargumente (etwa der zu suchende Text) und/oder Eingabedateien.
- Diese Konventionen sind im POSIX-Standard festgehalten.

- Zusätzlich zu dem zu suchenden Text soll es möglich sein. Optionen anzugeben (beides in Nachbildung des originalen *grep*-Kommandos):
  - Die Option „-n“ (*number*) soll die jeweilige Zeilennummer mit ausgeben.
  - Die Option „-v“ (*veto*) soll dazu führen, dass nur die Zeilen ausgegeben werden, die den Suchtext *nicht* enthalten.
- Die Optionen sollen kombinierbar sein, d.h. „-n“ und „-v“ können als zwei getrennte Parameter angegeben werden oder auch kombiniert, also etwa „-nv“ oder „-vn“.
- Der Konvention folgend soll „--“ die Optionen beenden. Bei dem Kommando „mygrep1 -n -- -1“ wird „-1“ nicht als die (nicht vorhandene) Option „1“ interpretiert, sondern als der Suchtext „-1“.

Suchkommando mit Optionen

mygrep1.c

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

int main(int argc, char *argv[]) {
   char *cmdname = *argv++; --argc; /* take command name */
   bool opt_v = false; /* option -v: print non-matching lines */
   bool opt_n = false; /* option -n: emit line numbers */

   /* process options ... */

   /* do the actual work */
   char line[256]; /* input line */
   int lineno = 0; /* current line number */
   while (fgets(line, sizeof line, stdin)) {
      lineno++;
      if (!strstr(line, pattern) == opt_v) {
         if (opt_n) printf("%d: ", lineno);
         fputs(line, stdout);
      }
   }
}
```

Suchkommando mit Optionen

mygrep1.c

```
/* process options */
for(; argc > 0 && **argv == '-'; argc--, argv++) {
   /* per convention we interpret "--" as end of options */
   if (argv[0][1] == '-' && argv[0][2] == 0) {
      argc--; argv++; break;
   }
   if (argv[0][1] == 0) {
      /* got just a "-" without anything following */
      fprintf(stderr, "%s: empty option\n", cmdname);
      return 1;
   }
   /* process individual options within an argument */
   for (char* s = *argv + 1; *s; s++) {
      switch (*s) {
         case 'v' : opt_v = true; break;
         case 'n' : opt_n = true; break;
         default:
            fprintf(stderr, "%s: illegal option '%c'\n", cmdname, *s);
            return 1;
      }
   }
}

/* just one remaining argument with the pattern is expected */
if (argc != 1) {
   fprintf(stderr, "Usage: %s [-nv] pattern\n", cmdname);
   return 1;
}
char* pattern = *argv++; --argc;
```

Unterstützung beliebig langer Zeilen

mygrep2.c

```
char* readline(FILE* fp) {
   int len = 32;
   char* cp = malloc(len);
   if (!cp) return 0;
   int i = 0;
   int ch;
   while ((ch = getc(fp)) != EOF && ch != '\n') {
      cp[i++] = ch;
      if (i == len) {
         /* double the allocated space */
         len *= 2;
         char* newcp = realloc(cp, len);
         if (!newcp) {
            free(cp);
            return 0;
         }
         cp = newcp;
      }
   }
   if (i == 0 && ch == EOF) {
      free(cp);
      return 0;
   }
   cp[i++] = 0;
   return realloc(cp, i); /* free unused space */
}
```

- Bislang waren die Optionen nur **bool**-wertig.
- Gelegentlich haben diese aber einen größeren Wertebereich, z.B. eine ganze Zahl oder ein Dateiname.
- *grep* kennt z.B. eine Option, die den jeweils anzugebenden Kontext spezifiziert (Zahl der zu zeigenden Zeilen davor und danach).
- Mit „-c 3“ sind beispielsweise drei Zeilen Kontext darzustellen.
- Konventionellerweise ist es aber auch zulässig, „-c3“ anzugeben oder in Kombination: „-nc3“.

Optionen mit Werten

mygrep3.c

```
for (char* s = *argv + 1; *s; s++) {
   switch (*s) {
      case 'c' :
         if (s[1]) {
            ++s;
         } else {
            ++argv; --argc;
            if (!argc) {
               fprintf(stderr,
                  "%s: argument for option 'c' missing\n", cmdname);
            }
            s = *argv;
         }
         /* pick argument from the rest of s[] */
         for (; *s; ++s) {
            if (!isdigit(*s)) {
               fprintf(stderr,
                  "%s: digits expected for option 'c'\n", cmdname);
               return 1;
            }
            context = context * 10 + *s - '0';
         }
         --s; /* break from outer for loop */
         break;
      case 'v' : opt_v = true; break;
      case 'n' : opt_n = true; break;
      default:
         fprintf(stderr, "%s: illegal option '%c'\n", cmdname, *s);
         return 1;
   }
}
```

- Es ist unerfreulich und fehleranfällig, die Kommandozeilenbearbeitung von Optionen „per Hand“ vorzunehmen.
- Es gibt daher im Rahmen des POSIX-Standards die Funktion *getopt*, die die Konventionen unterstützt.
- *getopt* erhält *argc* und *argv* (einschließlich dem Kommandonamen) und eine Optionsspezifikation, bestehend aus den Buchstaben der Optionennamen. Steht ein „:“ hinter einem Optionsbuchstaben, so erwartet die entsprechende Option einen Wert.
- Unsere *grep*-Nachimplementierung bräuchte also die Spezifikation „c:nv“.

Verarbeitung der Optionen mit *getopt*

mygrep4.c

```
char usage[] = "Usage: %s [-c context] [-nv] pattern\n";
/* external variables set by getopt() */
extern char* optarg;
extern int optind;
/* process options */
int option;
while ((option = getopt(argc, argv, "c:nv")) != -1) {
   switch (option) {
      case 'c':
         context = atoi(optarg); break;
      case 'n':
         opt_n = true; break;
      case 'v':
         opt_v = true; break;
      default:
         fprintf(stderr, usage, cmdname); return 1;
   }
}
argc -= optind; argv += optind; /* skip options processed by getopt() */
/* just one remaining argument with the pattern is expected */
if (argc != 1) {
   fprintf(stderr, usage, cmdname); return 1;
}
char* pattern = *argv++; --argc;
```

mygrep5.c

```
/* compile pattern */
regex_t regex; /* compiled regular expression */
unsigned int regex_flags = REG_NOSUB;
if (opt_i) {
   regex_flags |= REG_ICASE; /* ignore case */
}
if (opt_e) {
   regex_flags |= REG_EXTENDED; /* supported egrep syntax */
}
unsigned int regex_error = regcomp(&regex, pattern, regex_flags);
if (regex_error) {
   char errbuf[128];
   regerror(regex_error, &regex, errbuf, sizeof errbuf);
   fprintf(stderr, "%s: invalid regular expression: %s\n",
      cmdname, errbuf);
   return 1;
}
```

- Die POSIX-Bibliothek bietet auch eine Bibliothek für reguläre Ausdrücke an. Hier muss zunächst der reguläre Ausdruck in eine interne Datenstruktur übersetzt werden.

mygrep5.c

```
if ((regexec(&regex, line, 0, 0, 0) != 0) == opt_v) {
```

- Statt *strstr* wird dann *regexec* verwendet mit der zuvor einmal erstellten Datenstruktur.
- *regexec* liefert 0 zurück, wenn der reguläre Ausdruck für einen Teil der Zeichenkette zutrifft.
- *regexec* unterstützt auch das Extrahieren von Teilen einer Zeichenkette mit Hilfe von Klammern-Ausdrücken. Davon wird hier aber kein Gebrauch gemacht und deswegen sind zwei Parameter auf 0 gesetzt.