- Anders als in Java gibt es keine automatisierte Speicherverwaltung, die unbenötigte Speicherflächen automatisch freigibt.
- Entsprechend muss in C Speicher nicht nur explizit belegt, sondern auch explizit freigegeben werden.
- Dies ist recht fehleranfällig. Hinzu kommt, dass die Sicherheiten des Typsystems verlassen werden müssen.
- Zum Ausgleich dafür lässt sich eine Speicherverwaltung in C selbst schreiben.

void∗ *calloc*(*size_t nelem*, *size_t* elsize)
: Belegt Speicher für *nelem* Elemente der Größe *elsize* und initialisiert diesen mit 0. Im Erfolgsfall wird der Zeiger darauf geliefert, ansonsten 0.

void∗ *malloc*(*size_t size*)
: Belegt Speicher für ein Objekt, das *size* Bytes benötigt. Im Erfolgsfall wird der Zeiger darauf geliefert, ansonsten 0.

void free(**void**∗ *ptr*)
: Hier muss *ptr* auf eine zuvor belegte, jedoch noch nicht freigegebene Speicherfläche verweisen. Dann gibt *free* diese Fläche zur andersweitigen Nutzung wieder frei.

void∗ *realloc*(**void**∗ *ptr*, *size_t size*)
: Versucht, die Größe der Speicherfläche, auf die *ptr* verweist, auf *size* Bytes anzupassen. Im Erfolgsfalle wird ein Zeiger auf die (möglicherweise neue) Speicherfläche zurückgeliefert, ansonsten 0.

void∗ *aligned_alloc*(*size_t* alignment, *size_t* size)
: Neu eingeführt in C11, berücksichtigt Adresskanten.

Eine einfache lineare Liste

reverse.c

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

/* lineare Liste ganzer Zahlen */
typedef struct element {
   int i;
   struct element* next;
} element;

int main() {
   element* head = 0;
   int i;

   /* Zahlen einlesen und in der Liste
      in umgekehrter Reihenfolge ablegen */
   while ((scanf("%d", &i)) == 1) {
      element* last = (element*) calloc(1, sizeof(element));
      if (last == 0) {
         fprintf(stderr, "out of memory!\n"); exit(1);
      }
      last->i = i; last->next = head; head = last;
   }
   /* Zahlen aus der Liste wieder ausgeben */
   while (head != 0) {
      printf("%d\n", head->i);
      head = head->next;
   }
}
```

reverse.c

```
element* last = (element*) calloc(1, sizeof(element));
if (last == 0) {
   fprintf(stderr, "out of memory!\n"); exit(1);
}
```

- *calloc* wird hier darum gebeten, für ein Element der Größe **sizeof**(*element*) Speicher zu belegen.
- Das entspricht *element last* = **new** *element*() in Java.
- Falls der gewünschte Speicher nicht belegt werden kann, wird ein 0-Zeiger zurückgeliefert. Entsprechend ist in C immer ein anschließender Test auf 0 erforderlich.
- Wenn es klappt, wird durch *calloc* die Speicherfläche mit Nullen initialisiert.
- *calloc* liefert den generischen Zeiger **void**∗ zurück. Dieser ist zu allen anderen Zeigern kompatibel. Der Cast-Operator (*element*∗) macht diese Typkonvertierung hier explizit.

reverse.c

```
element* last = (element*) malloc(sizeof(element));
if (last == 0) {
   fprintf(stderr, "out of memory!\n"); exit(1);
}
```

- Alternativ kann auch *malloc* aufgerufen werden.
- *malloc* erwartet jedoch nur die Gesamtgröße der belegenden Speicherfläche in Bytes und unterlässt die Initialisierung.
- Der Inhalt des neuen Objekts ist deswegen im Erfolgsfall vollkommen uninitialisiert.

```
void* my_calloc(size_t nelem, size_t elsize) {
   void* ptr = calloc(nelem, elsize);
   if (ptr) return ptr; /* alles OK */
   /* Fehlerbehandlung: */
   fprintf(stderr, "out of memory -- aborting!\n");
   /* Termination mit core dump */
   abort();
}
```

- Die Behandlung des Falls, dass ein 0-Zeiger zurückgeliefert wird, sollte nie vergessen werden.
- Wem das zu mühsam erscheint, kann diese Überprüfung in eine separate Funktion auslagern wie *my_calloc* in diesem Fall.

```
char bigmem[8192]; /* hoffentlich ausreichend */
char* bigmem_free = bigmem;

void* alloc(size_t nbytes) {
   if (bigmem_free + nbytes >= bigmem + sizeof bigmem) {
      return 0; /* kein Speicher mehr da */
   }
   void* p = bigmem_free;
   bigmem_free += nbytes;
   return p;
}
```

- Eine triviale Speicherverwaltung, die nur Speicher aus einer Fläche abgibt, ohne freiwerdenden Speicher entgegennehmen zu können, ist einfach zu implementieren.
- Hier dient das große Array *bigmem* als Speicher-Reservoir.
- Der Zeiger *bigmem_free* zeigt auf den Bereich aus dem Array, der noch nicht vergeben ist. Alles davor wurde bereits vergeben. Zu Beginn wird der Zeiger auf den Anfang des großen Arrays gesetzt.

```
char bigmem[8192]; /* hoffentlich ausreichend */
char* bigmem_free = bigmem;

void* alloc(size_t nbytes) {
   if (bigmem_free + nbytes >= bigmem + sizeof bigmem) {
      return 0; /* kein Speicher mehr da */
   }
   void* p = bigmem_free;
   bigmem_free += nbytes;
   return p;
}
```

- Wenn es in der Speicherverwaltung darum geht, Adressarithmetik zu verwenden, dann wird **char**$*$ als Zeigertyp benutzt, da ein **char** die Größe eines Bytes hat, d.h. **sizeof**(**char**)$==1$.
- (Theoretisch legt der Standard nur fest, dass ein Byte mindestens 8 Bits hat. Prinzipiell kann ein Byte größer sein, aber der Standard definiert fest, dass ein Byte im Sinne des Standards von **char** repräsentiert wird und dass **sizeof**(**char**)$==1$ gilt.)
- Entsprechend ist **char** die kleinste adressierbare Größe.

badalign.c

```
char* cp = alloc(sizeof(char));
int* ip = alloc(sizeof(int));
if (cp && ip) {
   * cp = 'x'; * ip = 1;
} else {
   fprintf(stderr, "alloc failed\n");
}
```

- Was passiert, wenn wir über *alloc* unterschiedlich große Objekte anfordern?
- Dann zeigt *ip* auf einen ungeraden Wert!
- Manchen Plattformen macht das nichts aus, andere hingegen akzeptieren dies nicht:

```
clonard$ badalign
Bus Error (core dumped)
clonard$
```

- Manche Architekturen akzeptieren nicht, wenn größere Datentypen wie etwa **int** oder **double** an irgendwelchen ungeraden Adressen liegen.
- Die SPARC-Architektur beispielsweise besteht darauf, dass **int**-Variablen auf durch vier teilbaren Adressen liegen und **long long int** oder **double** auf durch acht teilbare Adressen.
- Andere Architekturen sind diesbezüglich großzügiger (wie etwa die x86-Plattform), aber mitunter ist die Zugriffszeit größer, wenn keine entsprechend ausgerichtete Adresse benutzt wird.
- Wenn es wegen einem Alignment-Fehler zu einem Absturz kommt, wird das als „Bus Error“ bezeichnet (mit Verweis auf den Systembus, der sich geweigert hat, auf ein Objekt an einer nicht ausgerichteten Adresse zuzugreifen).
- Eine Speicherverwaltung muss darauf Rücksicht nehmen.

alignment.c

```
#include <stdio.h>
#include <stdalign.h>

int main() {
   printf("alignment for char: %zd\n", alignof(char));
   printf("alignment for int: %zd\n", alignof(int));
   printf("alignment for long long int: %zd\n", alignof(long long int));
   printf("alignment for double: %zd\n", alignof(double));
   printf("alignment for double [10]: %zd\n", alignof(double [10]));
}
```

- Seit dem aktuellen Standard (C11) gibt es in C den **alignof**-Operator. Zu beachten ist, dass dieser den **#include** <stdalign.h>-Header benötigt.

```
thales$ ./alignment
alignment for char: 1
alignment for int: 4
alignment for long long int: 8
alignment for double: 8
alignment for double [10]: 8
thales$
```

```
void* alloc(size_t nbytes, size_t align) {
   char* p = bigmem_free;
   p = (char*) (((uintptr_t) p + align - 1) & ~(align - 1));
   if (p + nbytes >= bigmem + sizeof bigmem) {
      return 0; /* kein Speicher mehr da */
   }
   bigmem_free = p + nbytes;
   return p;
}
```

- Das Problem kann durch einen zusätzlichen Parameter mit der gewünschten Ausrichtung gelöst werden. Hier: *align*.
- Der Zeiger *p* wird dann auf die nächste durch *align* teilbare Adresse gesetzt unter der Annahme, dass *align* eine Zweierpotenz ist.

```
p = (char*) (((uintptr_t) p + align - 1) & ~(align - 1));
```

$(uintptr\_t)p$	konvertiere p in eine ganze Zahl
$align - 1$	setzt die n niedrigswertigen Bits, wobei $n = log_2(align)$
$\sim(align - 1)$	bildet das Komplement davon, d.h. alle Bits außer den n niedrigstwertigen sind gesetzt
$\& \sim(align - 1)$	blendet die n niedrigswertigen Bits weg
$(uintptr\_t)p + align - 1$	die Addition stellt sicher, dass das Resultat nicht kleiner wird durch das Wegblenden von Bits

Das setzt voraus, dass *align* eine Zweierpotenz ist. Davon ist bei Alignment-Größen immer auszugehen.

```
#define NEW(T) alloc(sizeof(T), alignof(T))

char* cp = NEW(char);      printf("cp =  %p\n", cp);
int* ip = NEW(int);        printf("ip =  %p\n", ip);
char* cp2 = NEW(char);     printf("cp2 = %p\n", cp2);
char* cp3 = NEW(char);     printf("cp3 = %p\n", cp3);
double* dp = NEW(double); printf("dp =  %p\n", dp);
```

```
thales$ goodalign
cp =  8049ba0
ip =  8049ba4
cp2 = 8049ba8
cp3 = 8049ba9
dp =  8049bb0
thales$
```

- *calloc*, *malloc* und *realloc* haben jedoch keine *align*-Parameter.
- In der Praxis wird die Größe des gewünschten Datentyps und die maximal vorkommende Alignment-Größe (typischerweise die von **double**) in Betracht gezogen.
- Beginnend ab C11 steht auch

  void∗ *aligned_alloc*(*size_t alignment*, *size_t size*);

  zur Verfügung. Hierbei muss *alignment* eine vom System tatsächlich verwendete Alignment-Größe sein und *size* ein Vielfaches von *alignment*.

- Wenn ein Prozess unter UNIX startet, wird zunächst nur Speicherplatz für den Programmtext (also den Maschinen-Code), die globalen Variablen, die Konstanten (etwa die von Zeichenketten) und einem Laufzeitstapel (Stack) angelegt.
- All dies liegt in einem sogenannten virtuellen Adressraum (typischerweise mit 32- oder 64-Bit-Adressen), den das Betriebssystem einrichtet.
- Die Betonung liegt auf virtuell, da die verwendeten Adressen nicht den physischen Adressen entsprechen, sondern dazwischen eine durch das Betriebssystem konfigurierte Abbildung durchgeführt wird.
- Diese Abbildung wird nicht für jedes einzelne Byte definiert, sondern für größere Einheiten, die Kacheln (*page*).

getpagesize.c

```
#include <stdio.h>
#include <unistd.h>

int main() {
   printf("page size = %d\n", getpagesize());
}
```

- Die Größe der Kacheln ist plattformabhängig und kann mit Hilfe des Systemaufrufs *getpagesize()* ermittelt werden.

```
thales$ uname -a
SunOS thales 5.10 Generic_147441-09 i86pc i386 i86pc
thales$ getpagesize
page size = 4096
thales$
```

```
clonard$ uname -a
SunOS clonard 5.10 Generic_144500-19 sun4u sparc SUNW,A70
clonard$ getpagesize
page size = 8192
clonard$
```

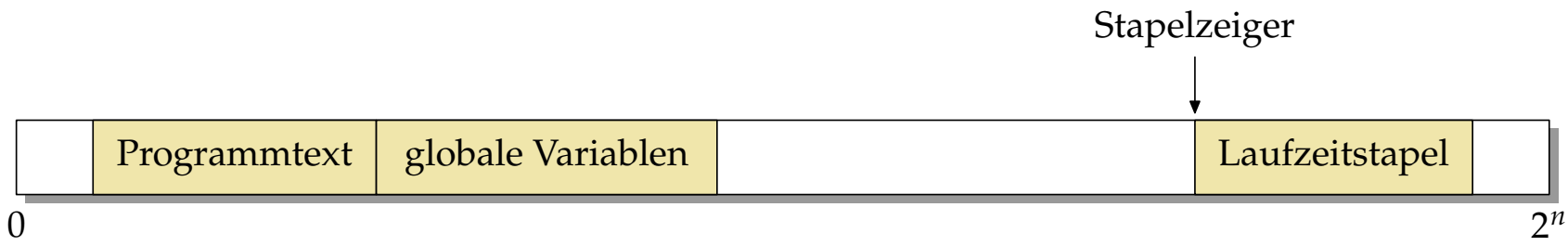
- Sei $[0, 2^n - 1]$ der virtuelle Adressraum, $P = 2^m$ die Größe einer Kachel und

  $$M : [0, 2^{n-m} - 1] \to \mathbb{N}_0$$

  die Funktion, die eine Kachelnummer in die korrespondierende physische Anfangsadresse abbildet.
- Dann lässt sich folgendermassen aus der virtuellen Adresse a_{virt} die zugehörige physische Adresse a_{phys} ermitteln:

  $$a_{phys} = M(a_{virt} \textbf{ div } P) + a_{virt} \textbf{ mod } P$$

- Die Funktion M wird von der zur Hardware gehörenden MMU (*memory management unit*) implementiert in Abhängigkeit von Tabellen, die das Betriebssystem konfigurieren kann.
- Für weite Teile des Adressraums bleibt M jedoch undefiniert. Ein Versuch, über einen entsprechenden Zeiger zuzugreifen, führt dann zu einem Abbruch des Programms (*segmentation violation*).



- Wie der Adressraum zu Beginn belegt wird, liegt in der Freiheit des Betriebssystems bzw. des *ld*.
- Fast immer bleibt die Adresse 0 unbelegt, damit Versuche, auf einen 0-Zeiger zuzugreifen, zu einem Fehler führen. Ebenso bleibt der ganz hohe Bereich bei 2^n frei.
- Der Programmtext und die globalen Variablen liegen normalerweise in der Nähe, sind aber unterschiedlich konfiguriert (*read-only/executable* und *read/write*).
- Der Laufzeitstapel (Stack) wird davon getrennt konfiguriert, damit er genügend Platz hat zu wachsen, typischerweise von hohen zu niedrigen Adressen.

end.c

```
#include <stdio.h>
extern void etext; extern void edata; extern void end;
int global[1000] = {1}; // some global initialized data
int main() { char local;
   printf("etext -> %p\n", &etext); printf("edata -> %p\n", &edata);
   printf("end -> %p\n", &end); printf("main -> %p\n", main);
   printf("global -> %p\n", global); printf("local -> %p\n", &local);
}
```

- Bei traditionellen *ld*-Konfigurationen werden die Symbole *etext*, *edata* und *end* definiert, die entsprechend auf das Ende des Programmtexts, das Ende der initialisierten Daten und das Ende der uninitialisierten Daten verweisen.

```
clonard$ ./end
etext -> 106f4
edata -> 2187c
end -> 218a8
main -> 105e8
global -> 208dc
local -> ffbff67f
clonard$
```

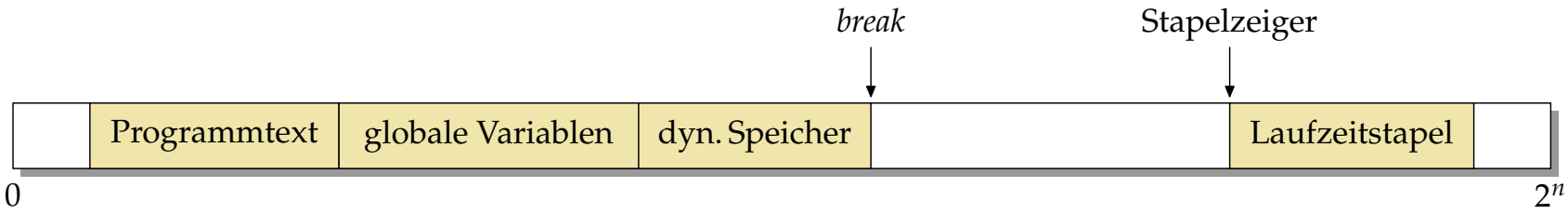
```
clonard$ pmap 26477
26477:  endpause
00010000       8K r-x--  /.../ws12/soft1/slides/examples/endpause
00020000       8K rwx--  /.../ws12/soft1/slides/examples/endpause
FF200000    1216K r-x--  /lib/libc.so.1
FF330000      40K rwx--  /lib/libc.so.1
FF33A000       8K rwx--  /lib/libc.so.1
FF360000       8K r-x--  /platform/sun4u-us3/lib/libc_psr.so.1
FF370000      24K rwx--    [ anon ]
FF380000       8K rwx--    [ anon ]
FF390000       8K rw---    [ anon ]
FF3A0000       8K rw---    [ anon ]
FF3B0000     232K r-x--  /lib/ld.so.1
FF3F0000       8K rwx--    [ anon ]
FF3FA000      16K rwx--  /lib/ld.so.1
FFBFC000      16K rw---    [ stack ]
 total      1608K
clonard$
```

- Solange ein Prozess noch läuft, kann auf Solaris mit Hilfe des *pmap*-Programms der virtuelle Adressraum aufgelistet werden.
- Links steht in Hex jeweils die Anfangsadresse, dann in dezimal die Zahl der belegten Kilobyte, dann die Zugriffsrechte (r = *read*, w = *write*, e = *executable*) und schließlich, sofern vorhanden, die in den Adressraum abgebildete Datei.

Zur vierten Spalte:

.../endpause	Das ausführbare Programm, das gestartet wurde.
/lib/libc.so.1	Dynamisch ladbare C-Bibliothek, „so“ steht dabei für *shared object*, die „1“ dient als Versionsnummer.
/lib/ld.so.1	Dynamisches Ladeprogramm (war notwendig, um */lib/libc.so.1* zu laden).
[stack]	Bis zu einem gewissen Limit automatisch wachsender Stack.
[anon]	Durch die Speicherverwaltung belegter Speicher.

- Jede dynamische Speicherverwaltung benötigt einen Weg, mehr Speicher vom Betriebssystem anzufordern und diesen in einen bislang ungenutzten Bereich des virtuellen Adressraums abzubilden.
- Dafür gibt es im POSIX-Standard zwei Systemaufrufe:
  - *sbrk* – der traditionelle Ansatz, einfach, aber nicht flexibel
  - *mmap* – sehr flexibel, aber auch etwas komplizierter
- Gearbeitet wird in jedem Fall mit ganzen Kacheln.
- Eine Rückgabe von Speicher scheitert normalerweise an der Fragmentierung. Bei C findet das normalerweise nicht statt, d.h. der belegte Speicher wächst dank der zunehmenden Fragmentierung langsam aber stetig.



- Der Break ist eine vom Betriebssystem verwaltete Adresse, die mit Hilfe der Systemaufrufe *brk* und *sbrk* manipuliert werden kann.
- *brk* spezifiziert die absolute Position des Break, *sbrk* verschiebt diese relativ.
- Zu Beginn zeigt der Break auf den Anfang des Heaps, konventionellerweise liegt dieser hinter den globalen Variablen.
- Durch das Verschieben des Breaks zu höheren Adressen kann dann Speicher belegt werden.

reverse.c

```
typedef struct buffer {
   struct buffer* next;
   size_t size; // size of the buffer pointed to by buf
   size_t left; // number of bytes left in buf
   char* buf; // points to free area behind struct buffer
      // [buf + left .. buf + size) is filled
} Buffer;
```

- *sbrk* liefert jeweils einen Zeiger auf den neuen Speicherbereich (ähnlich wie *malloc*, aber *sbrk* arbeitet nur mit Kacheln).
- Aufgabe ist es hier, beliebig lange Zeilen zu drehen. Gelöst wird dies durch eine lineare Liste von Puffern, die jeweils eine Kachel belegen.
- Um eine neu angelegte Kachel strukturiert verwenden zu können, wird hier eine Puffer-Struktur definiert.
- Die Struktur liegt jeweils am Anfang einer Kachel, der freie Rest wird für den Puffer-Inhalt, also den teilweisen Inhalt einer umzudrehenden Zeile belegt.

reverse.c

```
void print_buffer(Buffer* bufp) {
   while (bufp) {
      printf("%.*s", bufp->size - bufp->left, bufp->buf + bufp->left);
      bufp = bufp->next;
   }
}
```

- *print_buffer* geht durch die lineare Liste der Puffer, die für (eine vielleicht sehr lange) Zeile angelegt worden sind.
- *size* gibt jeweils an, wie groß der Puffer ist, *left* wieviel Bytes noch frei sind.
- Entsprechend ist der Bereich von *left* bis *size* gefüllt.

Reversierte Ausgabe von Zeilen

reverse.c

```
size_t pagesize = getpagesize();
Buffer* head = 0; // head of our linear list of buffers
Buffer* tail = 0; // tail of the list (first allocated)
Buffer* free = 0; // list of free buffers which can be recycled
char* cp;
int ch;
while ((ch = getchar()) != EOF) {
   if (ch == '\n') {
      // print buffer and release current chain of buffers
   } else {
      // allocate buffer, if necessary, and add ch to it
   }
}
if (head) print_buffer(head);
```

- Die Zeiger *head* und *tail* zeigen auf die lineare Liste von Puffern für die aktuelle Zeile.
- Der Zeiger *free* zeigt auf die lineare Liste ungenutzter Puffer, die erneut verwendet werden können.
- Bei Zeilentrennern und am Ende wird jeweils die Liste der Puffer mit *print_buffer* ausgegeben.

reverse.c

```
if (ch == '\n') {
   // print buffer and release current chain of buffers
   print_buffer(head); putchar('\n');
   if (tail) {
      // add them to the free list of buffers
      tail->next = free; free = head;
      head = 0; tail = 0;
   }
} else {
   // allocate buffer, if necessary, and add ch to it
}
```

- Bei einem Zeilentrenner wird die aktuelle lineare Liste der Puffer ausgegeben.
- Danach wird diese Liste freigegeben, indem sie in den Anfang der *free*-Liste eingefügt wird.

Reversierte Ausgabe von Zeilen

reverse.c

```
if (ch == '\n') {
   // print buffer and release current chain of buffers
} else {
   // allocate buffer, if necessary, and add ch to it
   if (!head || head->left == 0) {
      Buffer* bufp;
      if (free) {
         // take new buffer from our list of free buffers
         bufp = free; free = free->next;
      } else {
         // allocate a new buffer
         bufp = (Buffer*) sbrk(pagesize);
         if (bufp == (void*) -1) {
            perror("sbrk"); exit(1);
         }
      }
      bufp->next = head;
      bufp->size = pagesize - sizeof(struct buffer);
      bufp->left = bufp->size;
      bufp->buf = (char*)bufp + sizeof(struct buffer);
      head = bufp;
      if (!tail) tail = bufp;
      cp = bufp->buf + bufp->size;
   }
   *--cp = ch; --head->left;
}
```

reverse.c

```
// allocate a new buffer
bufp = (Buffer*) sbrk(pagesize);
if (bufp == (void*) -1) {
   perror("sbrk"); exit(1);
}
```

- *sbrk* verschiebt den Break um die angegebene Anzahl von Bytes. Diese sollte sinnvollerweise ein Vielfaches der Kachelgröße sein. Hier wird jeweils genau eine Kachel belegt.
- Im Fehlerfall liefert *sbrk* den Wert (**void***)−1 zurück (also nicht den Nullzeiger!).
- Wenn es geklappt hat, wird der *alte* Wert des Breaks geliefert. Das ist aber auch gleichzeitig der Beginn der neu belegten Speicherfläche, den wir danach nutzen können.

reverse.c

```
bufp->next = head;
bufp->size = pagesize - sizeof(struct buffer);
bufp->left = bufp->size;
bufp->buf = (char*)bufp + sizeof(struct buffer);
head = bufp;
if (!tail) tail = bufp;
cp = bufp->buf + bufp->size;
```

- Diese Zeilen initialisieren den neu angelegten Puffer und fügen diesen an den Anfang der linearen Liste ein.
- Die Puffer-Datenstruktur wird an den Anfang der Kachel gelegt. Der Rest der Kachel wird dem eigentlichen Puffer-Inhalt gewidmet, auf den *buf* zeigt.
- Die Position von *buf* wird mit Hilfe der Zeigerarithmetik bestimmt, wobei es entscheidend ist, dass zuerst *bufp* in einen **char**-Zeiger konvertiert wird, bevor die Größe der Puffer-Struktur addiert wird. Alternativ wäre aber auch
  $bufp{->}buf = (\mathbf{char}*)(bufp + 1)$
  denkbar gewesen.

- Das Beispiel zeigte, wie größere Speicherflächen (etwa Kacheln) beschafft werden und wie diese danach mit einzelnen Datenstrukturen belegt werden.
- Dies ist grundsätzlich in C möglich, wenn auf das Alignment geachtet wird. In diesem Fall war das trivial, weil der Anfang einer Kachel ausreichend ausgerichtet ist und hinter der Datenstruktur für den Puffer nur ein **char**-Array kam, das keine Alignment-Anforderungen hat.
- Eine Speicherverwaltung arbeitet ebenfalls mit größeren Speicherflächen, in denen sowohl die Verwaltung der Speicherflächen als auch die ausgegebenen Speicherbereiche integriert sind.