- 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 Speicherflächen zunächst nicht mit Datentypen verbunden sind. Sie müssen aufgeteilt, verwaltet und Zeiger müssen je nach Bedarf konvertiert werden. Entsprechend fehlt hier die Sicherheit des Typsystems.
- 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.