Dynamic storage

Vol. 2

Lecturers: Mirko Jantschke, Pascal Scholz

23. Januar 2019

Contents

Rückblick - Wichtige Funktionen

Ein kurzer Rückblick...

Besonders große oder dynamische Datenstrukturen können und sollten auf dem Heap angelegt werden. Der unäre Operator *sizeof* kann genutzt werden um die Größe von Objekten zu ermitteln.

Folgende Funktionen gibt es dafür in der *stdlib.h*:

- *malloc(size_t size)*
- *calloc(size_t nitems, size_t size)*
- *realloc(void *ptr, size_t size)*
- *free(void *ptr)*

malloc()

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

Die Funktion malloc() 'fragt' beim Betriebssystem nach Speicher in der Größe von *size* Bytes. Ist dies erfolgreich, wird ein Pointer auf den Anfang des Speicherbereiches zurückgegeben. Schlägt die Allokation fehl, wird ein *NULL*-pointer zurückgeben. Der Speicher wird nicht initialisiert!

```
1 /* Ein Beispiel */
2  int *eineZahl = malloc(sizeof(*eineZahl));
3  if(eineZahl != NULL) {
4     *eineZahl = 3;
5     /* Und noch viel mehr... */
6  }
```

calloc()

```
void *calloc(size_t nitems, size_t size);
```

Erfüllt die selbe Funktion wie *malloc()*, aber der Speicher wird mit dem Wert 0 initialisiert.

```
1 /* Ein Beispiel */
2  int *eineZahl = calloc(sizeof(*eineZahl));
3  if(eineZahl != NULL) {
4     printf("Ausgabe: %d\n", *eineZahl); // Ausgabe: 0
5     /* Und noch viel mehr... */
6  }
```

realloc()

```
void *realloc(void *ptr, size_t size);
```

Erfüllt die selbe Funktion wie *malloc()*, wenn *ptr == NULL* ist, sonst wird neuer Speicher in Größe von *size* Bytes allokiert. Auf *ptr* wird free aufgeführt, also nicht mehr verwenden!

```
1 /* Ein Beispiel */
2  int *einArray = calloc(sizeof(*eineZahl));
3  int *neu = realloc(eineZahl, 2 * sizeof(*neu));
4  if(neu != NULL) {
5     einArray = neu;      // Jetzt neu mit Platz fuer 2 integer
6     einArray[0] = 1;
7     einArray[1] = 2;
8     /* Und noch viel mehr... */
9  }
```

free()

```
void free(void *ptr);
```

Gibt den mit *malloc()*, *calloc()* oder *realloc()* auf dem Heap reservierten Speicher wieder frei. Der pointer *ptr* sollte nicht mehr verwendet werden.

```
1 /* Ein Beispiel */
2  //Speicher wird allokiert
3  int *eineZahl = calloc(sizeof(*eineZahl));
4  if(eine != NULL) {
5     /* Viel Code... */
6     free(eineZahl); //Speicher wird freigegeben
7  }
8  return 0;
```

Die verkettete Liste - Ein Beispiel

Was wir vorhaben

Nun wollen wir das Ganze an einem Beispiel ausprobieren.

Das Ziel: Wir wollen eine Datenstruktur bauen, mit der wir problemlos beliebig viele Zahlen speichern können.

Das Ganze werden wir nun Schritt für Schritt zusammen erarbeiten.

Linked List - Konzept

Eine verlinkte List, bzw linked list, speichert neben einem Element auch einen Verweis auf das nächste Element. Merkt man sich das Anfangselement, kann man alle anderen Elemente erreichen. Verliert man das erste Element, verliert man die gesamte Liste!

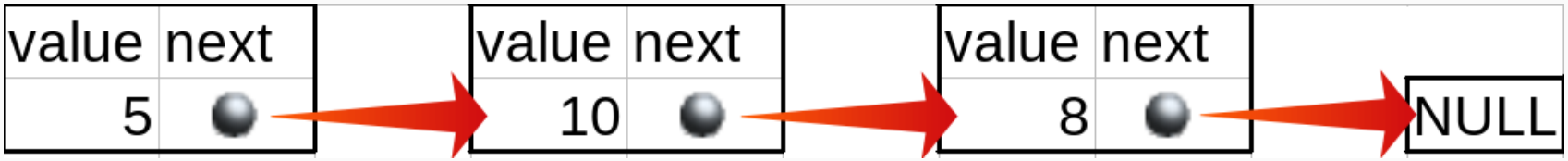

Abbildung 1: Beweis, dass LibreOffice Calc ein vollwertiges Bildbearbeitungprogramm ist.

Schritt 1 - Die Datenstruktur

Schritt 1 - Die Datenstruktur

Wir benötigen eine Datenstruktur, welche einen Pointer auf das nächste Element der Liste und eine Zahl speichern kann.

Schritt 1 - Die Datenstruktur

Wir benötigen eine Datenstruktur, welche einen Pointer auf das nächste Element der Liste und eine Zahl speichern kann.

Wir benutzen ein Struct!

```
struct list_elemnt {
    /* ?? */
};
```

Schritt 1 - Die fertige Datenstruktur

Diese Struktur könnne wir verwenden:

```
1 struct list_element {
2     int value;
3     struct list_element *next;
4 };
```

Schritt 2 - Verwenden der Datenstruktur und Einfügen

Schritt 2 - Wir erzeugen 2 Elemente zum Testen

Wir können nun zwei Elemente unserer Struktur erzeugen um diese zu testen. Diese legen wir auf dem Heap an, da unsere Liste später einmal dynamisch wachsen soll.

```
1 int main(void) {
2     struct list_element *list = malloc(sizeof(*a));
3     struct list_element *newEntry = malloc(sizeof(*b));
4     return 0;
5 }
```

Schritt 2 - Wertezuweisung

Diesen sollten Werte zugewiesen werden..

```
1 int main(void) {
2     struct list_element *list;
3     /* Zuweisung */
4     return 0;
5 }
```

Schritt 2 - Wertezuweisung

Diesen sollten nun Werte zugewiesen werden...

```
1 int main(void) {
2     struct list_element *list;
3     struct list_element *newEntry;
4     list->value = 10;
5     list->value = NULL;
6     /* Analog fuer newEntry */
7     return 0;
8 }
```

Schritt 2 - Wertezuweisung

Jetzt können wir beide Elemente verketten.

Doch wie?

Schritt 2 - Wertezuweisung

Jetzt können wir beide Elemente verketten.

... indem wir den Pointer setzen!

```
1 int main(void) {
2     struct list_element *list;
3     struct list_element *newEntry;
4     /* Code */
5     list->next = newEntry;
6     /* Code */
7 }
```

Schritt 3 - Die Liste ausgeben

Schritt 3 - Die Ausgabe

Nun haben wir eine Liste mit zwei Elementen. Wir können diese natürlich nun wie folgt jeweils ausgeben:

```c
int main(void) {
    struct list_element *list;
    struct list_element *newEntry;
    /* Code */
    printf("Element 1: %d", list->value);
    printf("Element 2: %d", list->next->value);
    /* Code */
}
```

Wenn wir aber mehr als zwei Elemente in unserer Liste haben und das für jedes Element machen müssten, würde diese Lösung schnell unpraktisch werden.

Schritt 3 - Die Schleife...

Schlauer ist es dafür eine Schleife anzulegen:

```c
int main(void) {
    /* Code */
    /* Dies ist eine Luecke */
    while (/* Luecke */) {
        printf("Element: %d", /* Luecke */);
        /* Luecke */
    }
    /* Code */
}
```

Schritt 3 - Die Schleife

Schlauer ist es dafür eine Schleife anzulegen:

```c
int main(void) {
    /* Code */
    struct list_element *elem = list;
    while (/* Luecke */) {
        printf("Element: %d", /* Luecke */);
        /* Luecke */
    }
    /* Code */
}
```

Schritt 3 - Die Schleife...

Schlauer ist es dafür eine Schleife anzulegen:

```c
int main(void) {
    /* Code */
    struct list_element *elem = list;
    while (elem != NULL) {
        printf("Element: %d", /* Luecke */);
        /* Luecke */
    }
    /* Code */
}
```

Schritt 3 - Die Schleife...

Schlauer ist es dafür eine Schleife anzulegen:

```c
int main(void) {
    /* Code */
    struct list_element *elem = list;
    while (elem != NULL) {
        printf("Element: %d", /* Luecke */);
        elem = elem->next;
    }
    /* Code */
}
```

Schritt 3 - Die Schleife...

Schlauer ist es dafür eine Schleife anzulegen:

```c
int main(void) {
    /* Code */
    struct list_element *elem = list;
    while (elem != NULL) {
        printf("Element: %d", elem->value);
        elem = elem->next;
    }
    /* Code */
}
```

Wenn wir unsere Liste mehrmals ausgeben wollen, bietet es sich an, diese Schleife in eine Funktion auszulagern.

Schritt 3 - ...ausgelagert in eine Funktion

```c
void printList(struct list_element *begin) {
    while (begin != NULL) {
        printf("Element: %d", begin->value);
        begin = begin->next;
    }
}

int main(void) {
    struct list_element *list;
    /* Code */
    printList(list);
    /* Code */
}
```

Der Pointer *begin* ist dabei eine lokale Kopie der übergebenen Adresse. Solange wir diesen nicht derefierzieren (mit *begin), wird *list* nicht überschrieben.

Schritt 4 - Auslagern des Einfügens

Schritt 4 - Das Einfügen wird...

Nun möchten wir beim Einfügen eines neuen Elementes dieses nicht immer von Hand verlinken.

```
1 int main(void) {
2     /* Code */
3     struct list_element *last = /* Das letzte Element von list*/
      ;
4     struct list_element *new = malloc(sizeof(*new));
5     new->value = 24;
6     new->next  = NULL;
7     last->next = new;
8     /* Code */
9 }
```

Schritt 4 - ...ebenfalls ausgelagert

Das könnte wie folgt aussehen:

```c
void append(struct list_element *begin, int value) {
    /* Code */
}

int main(void) {
    /* Code */
    struct list_element *list = malloc(sizeof(*list));
    /* Code*/
    append(list, 24);
    /* Code */
}
```

Aufgaben

Vervollstädigung append()

Die Funktion *append()* soll ein neues *list_element* am Ende der Liste einfügen. Der Wert, der im neuen Element gespeichert wird, soll *value* sein.

```
1 void append(struct list_element *begin, int value) {
2     /* Code */
3 }
```

Zusatz: Verändere die Datenstruktur so, dass das Einfügen am Ende schneller wird. Eventuell sind dafür auch Anpassungen an den anderen Funktionen nötig.

Einfügen nach Index

Die Funktion *insert()* soll ein neues *list_element* an der Stelle *index* der Liste einfügen. Der Wert, der im neuen Element gespeichert wird, soll *value* sein.

```
1 void insert(struct list_element *begin, int value, size_t index)
       {
2     /* Code */
3 }
```

Zum Beispiel soll in die Liste 1-2-3-4 und *index* == 2 und *value* == 24 der Wert 24 zwischen die 1 und die 2 eingefügt werden, also 1-24-2-3-4.
ACHTUNG: Es müssen Randfälle beachtet werden.