6.4.A Extras/Dynamische Speicherverwaltung – Arbeitsblatt

Aufgabe 1: `malloc()` statt Array

Schreiben Sie ein Hello-World-Programm mit dynamischem Speicher! Der Speicher für den auszugebenden String soll per `malloc()` geholt werden. Anschließend soll der Text mit `strcpy()` in den neuen Speicherbereich gefüllt werden. Schließlich soll er ausgegeben werden.

- Welche Header-Dateien muss man einbinden, und wofür muss man sie einbinden?
- Wieviel Speicher muss man für den String "Hello, World!\n" allozieren (holen)?
- Wenn man `malloc()` benutzt, muss man das Programm um zwei weitere Anweisungen ergänzen. Welche sind das?
- Schreiben Sie das Programm. Es soll ohne Warnungen übersetzt werden und allen gehaltenen Speicher wieder zurückgeben.

Aufgabe 2: Array mit variabler Länge, einmal VLA mit, einmal mit `malloc`

In dieser Aufgabe soll der Benutzer gefragt werden, wie viele Werte vom Typ `double` er braucht. Anschließend soll ein Speicherbereich der entsprechenden Größe angelegt werden. Zum Schluss soll der Bereich mit Null-Bytes gefüllt werden.

- Mit welcher Funktion aus der Standard-Bibliothek könnte das Füllen mit Null-Bytes geschehen?
- Lösen Sie das Problem mit VLAs! Wie groß kann der Speicherbereich maximal sein, und was passiert, wenn man ein zu großes Array haben möchte?
- Lösen Sie das Problem mit dynamischem Speicher, also durch Aufruf von `malloc()`! Wie groß kann der Speicherbereich jetzt maximal sein, und was passiert, wenn man zu viel Speicher anfordert?

Aufgabe 3: Zusatzaufgabe: Eingabe einer beliebig langen Zeichenkette

In der Standardbibliothek fehlt leider eine Funktion, die eine beliebig lange Zeichenkette einlesen kann. Hier ist ein Prototyp:

```
1 char *mgets(void);
```

Der Aufruf sähe so aus:

```
1 char *p;  
2 p=mgets();
```

- Was muss man tun, wenn man die Zeichenkette nicht mehr braucht?
- Erstellen Sie die Funktion!
- Testen Sie die Funktion mit einem Programm, das eine beliebig lange Zeichenkette von der Tastatur einliest und rückwärts wieder ausgibt!

Aufgabe 4: Strategien der Speicherverwaltung I

Je nach C-Compiler, Standardbibliothek und Betriebssystem kann die dynamische Speicherverwaltung anders organisiert sein. Auf einem Mikrocontrollersystem kann sie eventuell sogar ganz fehlen (weil sie dort nicht gebraucht wird). Es ist jedoch ganz hilfreich, wenn man ein paar Rahmendaten weiß. Zur Hilfe dient das unten abgedruckte Programm. Dort werden zwei Speicherbereiche gleicher Größe mit `malloc()` geholt.

- Führen Sie das Programm aus, geben Sie auf die Frage nach der Speichergröße eine kleine Zahl ein. Wie liegen die Adressen der Stack-Variablen `u` und der beiden Heap-Variablen `*p` und `*q` relativ zueinander?
- Geben Sie diesmal an, dass Sie ein Byte brauchen. Wie weit liegen die Adressen von `*p` und `*q` auseinander?
- Geben Sie nun einen anderen Wert ein. Wie weit liegen die Adressen von `*p` und `*q` auseinander? Finden Sie eine Regelmäßigkeit!

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     char *p, *q;
7     unsigned long sz;
8     char u; /* unbenutzt */
9
10    printf("Variable u liegt an Adresse %u\n", &u);
11    printf("Gewuenschte Speichermenge: ");
12    scanf("%lu", &sz);
13    p=malloc(sz);
14    q=malloc(1);
15    printf("p=%u, q=%u, q-p=%u\n", p, q, q-p);
16    free(p); free(q);
17    return 0;
18 }
```

Aufgabe 5: Strategien der Speicherverwaltung II

Nun sollen Sie folgende Fälle für Ihr System untersuchen:

- `p=malloc(1); p[1]='x'; free(p); p=malloc(1);` – Was passiert?
- `p=malloc(1); free(p); q=malloc(1); free(q);` – Gesucht: Adressen
- `p=malloc(1); q=malloc(1); free(p); free(q); p=malloc(1);` – Gesucht: Adressen
- `p=malloc(1); q=malloc(1); free(q); free(p); p=malloc(1);` – Gesucht: Adressen
- `p=malloc(1); free(p); free(p);` – Was passiert?