

C/C++ Übungsblatt 5 (Block 1)

Prof. Dr. Klaus Obermayer und Mitarbeiter

Dynamischer Speicher und Rekursion

Verfügbar ab:	21.11.2022
Abgabe bis:	28.11.2022

Aufgabe 1: Vektorimplementierung

7 Punkte

Schreiben Sie ein Programm, einen dynamisch großen Speicher für beliebig lange Zeichenketten bereit stellt. Das Programm soll dazu ein Array von Strings, also Array von `char`-Pointern, mit dynamischer Länge implementieren. Nutzen Sie zur Umsetzung dynamische Speicherverwaltung, also Heap-Speicher.

Nutzen Sie die globale Variable vom Typ `char **storage`. Der `char **storage` ist ein Pointer auf einen Speicherbereich (ein Array), an dem beliebig viele `char *` (also `char`-Pointer) liegen. Nutzen Sie in den zu implementierenden Funktionen die im Tutorium kennen gelernten Funktionen zur Allokierung von dynamischem Speicher.

Zur Übersicht:

- Der `char **storage` zeigt auf einen Speicherbereich (bzw. Array), in dem so viele Variablen vom Typ `char-Pointer` aufgenommen werden, wie Sie Stringreferenzen speichern wollen.
- All diese `char-Pointer`-Variablen speichern die Adresse eines Strings

Nutzen Sie weiterhin die globale Variable `capacity` vom Typ `unsigned short` an, die die Kapazität (also die Zahl an `char-Pointern`, die in `storage` aufgenommen werden können) speichert.

Nun sollen folgende Funktionen implementiert werden:

- `void vector(unsigned short size)`: Wird zum initialisieren benutzt.
Legt Platz für `size` C-Strings an und initialisiert `capacity`. Alle `size` Stringreferenzen sollen initial einen Null-Pointer beinhalten, um das Arrayelement als leer zu markieren.
- `void push(char *string)`: Fügt am ersten unbenutzten Platz im `storage` die Stringreferenz `string` hinzu. Existiert kein freier Platz mehr, soll ein neuer `storage` mit doppelter Kapazität angelegt werden.
 - Verdoppeln Sie `capacity` und allokalieren Sie neuen Speicher
 - Kopieren Sie alle Elemente des alten Arrays in das neue Array
 - Fügen Sie die neue Stringreferenz `string` hinzu, für den bis eben noch kein Platz war
 - Stellen Sie sicher, dass die neuen leeren Felder des Arrays einen Null-Pointer beinhalten
 - Geben Sie den Speicher des alten Arrays frei
 - Speichern Sie das neue Array wieder in `storage`
- `char* at(unsigned short index)`: Liefert das String-Element am Index `i` zurück. Falls der Index außerhalb des Arrays liegt, soll ein Null-Pointer zurückgegeben werden.

- `void set(unsigned short index, char *string)`: Setzt den String am Index `i` auf `string`. Falls der Index außerhalb der Grenzen liegt, soll nichts passieren.
- `unsigned short size(void)`: Gibt die Anzahl der String-Elemente in dem Array zurück. Einträge, die aus dem Null-Pointer bestehen, sollen nicht mitgezählt werden.

War es zu einem beliebigen Zeitpunkt nicht möglich, genügend Speicher zu allokalieren, soll das Programm mit einer aussagekräftigen Fehlermeldung beendet werden.

Hinweis: Das Testprogramm legt Stringlitterale (im Read-Only-Speicher) an. Diese Referenzen (also char-Pointer) werden dann den zu implementierenden Funktionen übergeben, um Ihre Implementierung zu testen.

Hinweis: Wird ein Null-Pointer mit `%s` der `printf`-Funktion übergeben, führt dies zur Ausgabe `(null)` auf der Konsole.

Nutzen Sie die Vorgabe (auch auf ISIS verfügbar):

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char **storage;
5 unsigned short capacity;
6
7 void vector(unsigned short size)
8 {
9     // ... Hier Ihr Code ...
10 }
11
12 void push(char *string)
13 {
14     // ... Hier Ihr Code ...
15 }
16
17 char* at(unsigned short index)
18 {
19     // ... Hier Ihr Code ...
20 }
21
22 void set(unsigned short index, char *string)
23 {
24     // ... Hier Ihr Code ...
25 }
26
27 unsigned short size(void)
28 {
29     // ... Hier Ihr Code ...
30 }
31
32 int main(void)
33 {
34     // Erzeuge Test-Vector
35     vector(1);
36
37     // Füge 7 Strings hinten an
38     push("Wer");
39     push("Johannes");
40     push("Maria");
```

```
41 push("Olaf");
42 push("hat");
43 push("ein");
44 push("Problem");
45
46 // setze manuell 3 Elemente
47 set(1, "das");
48 set(2, "Anna");
49 set(3, "kann,");
50
51 // setze ein ungueltiges Element
52 set(77, "Haus");
53
54 // loesche drittes, sechstes und siebtes Element
55 set(2, 0);
56 set(5, 0);
57 set(6, 0);
58
59 // speichere neue Elemente
60 push("lesen");
61 push("alles");
62 push("richtig");
63 push("gemacht");
64 push(":)");
65
66 // Gebe Test-Vektor aus
67 for (int i = 0; i < capacity; ++i){
68     printf("%s ", at(i));
69 }
70 printf("\nInsgesamt %hu Eintraege.\n", size());
71 }
```

Aufgabe 2: Rekursive Folgen

3 Punkte

Implementieren Sie ein Programm, dass von folgenden Rekursiven Folgen ein bestimmtes Folgeglied berechnet. Das Programm soll dazu folgende Funktionen beinhalten:

- `int main(void)`
 1. – Ruft die Funktion `int rec1(int)` mit angemessenem Wert auf
 - Lässt sich das erste Folgeglied mit einem Wert über 100 zurückgeben
 - Gibt das berechnete Folgeglied auf der Konsole aus
 2. – Ruft die Funktion `int rec2(int)` mit angemessenem Wert auf
 - Lässt sich das 12. Folgeglied zurückgeben
 - Gibt das berechnete Folgeglied aus
- `int rec1(int):`
 - Berechnung der Folge `rec1`:
$$a_1 = 0$$
$$a_n = 3 * a_{n-1} + 1$$
 - Beendet die Berechnung, wenn a_n den Wert 100 übersteigt.
- `int rec2(int):`
 - Berechnung der Folge `rec2`:
$$a_1 = 0$$

$$a_n = 2 * a_{n-1} + 1 \text{ wenn } n \text{ ungerade}$$

$$a_n = 2 * a_{n-1} \text{ wenn } n \text{ gerade}$$

- Beendet die Berechnung, nachdem a_{12} vollständig berechnet wurde.

Hinweis: Die Berechnungen sind unbedingt rekursiv durchzuführen. Lösungen mit iterativem Vorgehen werden mit 0 Punkten gewertet.