Technische Universität Berlin **Neuronale Informationsverarbeitung**

C/C++ Übungsblatt 5 (Block 1)

Prof. Dr. Klaus Obermayer und Mitarbeiter

Dynamischer Speicher und Rekursion

Erinnerung: Bitte denken Sie an die Bearbeitung des ersten Block-Tests.

Verfügbar ab:	**07.12.2020**
Abgabe bis:	**14.12.2020**

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 initialisert `capacity`. Alle `size` Stringreferenzen sollen initial einen Null-Pointer beinhalten, um das Arraylement 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 ein neuer `storage` mit doppelter Kapazität angelegt werden.
  - Verdoppeln Sie `capacity` und allokieren 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 allokieren, soll das Programm mit einer aussagekräftigen Fehlermeldung beendet werden.

Hinweis: Das Testprogramm legt Stringliterale (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):

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

char **storage;
unsigned short capacity;

void vector(unsigned short size)
{
    // ... Hier Ihr Code ...
}

void push(char *string)
{
    // ... Hier Ihr Code ...
}

char* at(unsigned short index)
{
    // ... Hier Ihr Code ...
}

void set(unsigned short index, char *string)
{
    // ... Hier Ihr Code ...
}

unsigned short size(void)
{
    // ... Hier Ihr Code ...
}

int main(void)
{
    // Erzeuge Test-Vector
    vector(3);

    // Fuege 4 Strings hinten an
    push("Anton");
    push("Berta");
    push("Caesar");
```

```
41      push("schon");
42      push("aus");
43
44      // setze erste 3 Elemente
45      set(0, "Das");
46      set(1, "sieht");
47      set(2, "Dora");
48
49      // setze ein ungueltiges Element
50      set(100, "Friedrich");
51
52      // loesche zweites und viertes Element
53      set(2, 0);
54      set(4, 0);
55
56      // speichere neue Elemente
57      push("doch");
58      push("sehr");
59      push("gut");
60      push("aus");
61      push(":)");
62
63      // Gebe Test-Vektor aus
64      for (int i = 0; i < capacity; ++i){
65          printf("%s ", at(i));
66      }
67      printf("\nInsgesamt %hu Eintraege.\n", size());
68  }
```

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 200 zurückgeben
     – Gibt das berechnete Folgeglied auf der Konsole aus
  2. – Ruft die Funktion `int rec2(int)` mit angemessenem Wert auf
     – Lässt sich das 10. Folgeglied zurückgeben
     – Gibt das berechnete Folgeglied aus
- `int rec1(int)`:
  - Berechnung der Folge `rec1`:
    $a_1 = 0$
    $a_n = 3 * a_{n-1} + 2$
  - Beendet die Berechnung, wenn a_n den Wert 200 übersteigt.
- `int rec2(int)`:
  - Berechnung der Folge `rec2`:
    $a_1 = 1$
    $a_n = 2 * a_{n-1}$ wenn n ungerade
    $a_n = 2 * a_{n-1} - 1$ wenn n gerade

- Beendet die Berechnung, nachdem a_{10} vollständig berechnet wurde.

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