S. Kurtz, T. Schwabe,
G. Gonnella, D. Willrodt

Universität Hamburg
Zentrum für Bioinformatik

Programmierung für Naturwissenschaften
Sommersemester 2015
Übungen zur Vorlesung: Ausgabe am 16.04.2015

Punkteverteilung: Aufgabe 3.1-2: 2 Punkte, Aufgabe 3.3: 6 Punkte

Abgabe bis zum 22.04.2015, 10:00 Uhr.

Aufgabe 3.1 Makros können in C-Programmen nützliche Werkzeuge darstellen, um Konstanten oder einfache Funktionen zu ersetzen.

Die Datei `macro_tests.c` enthält einige Beispiele von einfachen Makros sowie deren Anwendung. Allerdings sind einige Makros nicht optimal definiert und produzieren fehlerhafte Ergebnisse. Identifizieren Sie diese Fehler und ändern Sie die Makros so ab, dass alle Funktionen die gewünschten Ergebnisse liefern. Ändern Sie dazu nur die Makros, nicht den restlichen Code.

In STiNE finden Sie wie gewohnt ein Makefile mit einem Test für Ihren Code.

Aufgabe 3.2 Schreiben Sie ein Programm, das für ein gegebenes Alphabet (d.h. eine Menge von Zeichen) und eine Länge k alle möglichen k-Worte über dieses Alphabet in lexikographisch sortierter Reihenfolge ausgibt.

Hinweis: Ein k-Wort ist ein Wort der Länge k.

Die Ausgabe in lexikographischer Reihenfolge soll von der Reihenfolge der Zeichen im gegebenen Alphabet abhängen.

Beispiele:

`./enumkmers.x ABC 2`

liefert die folgende Ausgabe:

```
AA
AB
AC
BA
BB
BC
CA
CB
CC
```

`./enumkmers.x ACB 2`

liefert die folgende Ausgabe:

```
AA
AC
AB
CA
CC
CB
BA
BC
BB
```

In STiNE finden Sie ein Makefile und ein Testscript, dass korrekt laufen sollte. (`make test`)

Aufgabe 3.3 Bei der dynamischen Speicherverwaltung mit `malloc` und `realloc` gibt es keine für den Benutzer zugängliche „Buchführung“ über die bereits allokierten Speicherbereiche. In größeren Programmen ist jedoch eine solche Buchführung sinnvoll. Man möchte z.B. wissen, ob man alle allokierten Speicherblöcke auch explizit wieder freigegeben hat, oder was die maximale Gesamtgröße

aller allokierten Speicherblöcke ist. In dieser Aufgabe sollen Sie in der Programmiersprache C ein Modul `memmanage.c` entwickeln, das die Buchführung über Speicherblöcke erlaubt. Der Speicher wird mit `realloc` allokiert. `realloc` wird jedoch nicht mehr direkt aufgerufen, sondern indirekt von den Funktionen, die der Memory-Manager zur Verfügung stellt. Das zu implementierende Modul basiert auf der folgenden Header Datei `memmanage.h`, die auch die vordefinierten Funktionsköpfe enthält.

```
#ifndef MEMMANAGE_H
#define MEMMANAGE_H

typedef struct MMspaceblock MMspaceblock;
typedef struct MMspacetable MMspacetable;

MMspacetable* mem_man_new(unsigned long numberofblocks);
void *mem_man_alloc(MMspacetable *st, char *file, unsigned long line,
                    void *ptr, unsigned long size, unsigned long number);
void mem_man_delete_ptr(MMspacetable *st, char *file, unsigned long line,
                        void *ptr);
void mem_man_info(const MMspacetable *st);
void mem_man_check(const MMspacetable *st);
void mem_man_delete(MMspacetable *st);

#endif
```

Die Information zu einem Speicherblock wird in einer Struktur vom Typ `MMspaceblock` abgelegt. Diese enthält den Zeiger auf einen von `realloc` allokierten Speicherblock. Außerdem sind darin noch die Größe einer jeden Zelle des Speicherblocks, sowie die Anzahl der Zellen des Speicherblocks festgehalten. Schließlich wird noch vermerkt, in welcher Zeile des Programms und in welcher Datei die Speicheranforderung erfolgte. Damit lassen sich leicht „space leaks“ lokalisieren, d.h. man kann herausfinden, an welcher Stelle des Programms ein Speicherblock angelegt wurde, der nicht freigegeben wird. Die Funktionen sollen folgendes leisten:

- Die Funktion `mem_man_new` allokiert mit `malloc` eine Speichertabelle mit der angegebenen Anzahl von Speicherblöcken, und initialisiert diese.
- Die Funktion `mem_man_alloc` allokiert mit `realloc` einen neuen Speicherblock (`ptr` ist gleich `NULL`) bzw. ändert die Größe eines bereits allokierten Speicherblocks (`ptr` ungleich `NULL`). Dabei wird die nötige Information über den betroffenen Speicherblock in der Speichertabelle vermerkt. Beachten Sie, dass `malloc(n)` äquivalent ist zu `realloc(NULL,n)`, d.h. man kann auf `malloc` verzichten. `size` gibt an, welchen Speicherbedarf der Typ des Zeigers hat, d.h. wie groß eine Zelle des Speicherblocks ist. `number` gibt an, für wieviele Elemente Speicher allokiert werden soll (z.B. bei einem Array). Das Produkt aus `size` und `number` gibt demnach die Größe des insgesamt für `ptr` zu allokierenden Speichers (in bytes) an. `file` und `line` gibt jeweils den Namen der Quelldatei an und die Zeilennummer, in der `mem_man_alloc` aufgerufen wurde. `mem_man_alloc` gibt als Rückgabewert die Adresse des allokierten Speicherblocks zurück. Über diese kann der Speicherblock eindeutig identifiziert werden.
- Die Funktion `mem_man_delete_ptr` gibt den für `ptr` allokierten Speicher mit Hilfe der Standardfunktion `free` wieder frei und vermerkt das in der Speichertabelle. Wurde der Zeiger `ptr` vorher nicht mit `mem_man_alloc` allokiert, d.h. gibt es keinen Eintrag zu diesem Zeiger in der Speichertabelle, so gibt `mem_man_delete_ptr` eine Fehlermeldung aus. Die Parameter `file` und `line` geben an, in welcher Quelldatei und in welcher Zeile dieser Datei `mem_man_delete_ptr` aufgerufen wurde.

- Die Funktion `mem_man_info` gibt für alle allokierten Speicherblöcke relevante Informationen aus. Das Format dazu muss dem Beispiel unten entsprechen.
- Die Funktion `mem_man_check` prüft, ob in der Speichertabelle noch nicht freigegebene Speicherblöcke vermerkt sind. Falls es solche gibt, wird für den ersten nicht freigegebenen Speicherblock ausgegeben, in welcher Zeile und in welcher Quelldatei dieser Block allokiert wurde (siehe Beispiel unten). Die Funktion bricht dann das Programm mit `EXIT_FAILURE` ab.
- Die Funktion `mem_man_delete` gibt den für die gesamte Speichertabelle allokierten Speicher wieder frei.

Beachten Sie, dass die Größe der Speichertabelle durch den Aufruf von `mem_man_new` vor dem ersten Aufruf der anderen Funktionen festgelegt wird. Die Funktionen `mem_man_alloc` und `mem_man_delete_ptr` sollen vom Programm nicht direkt, sondern nur über Makros aufgerufen werden. Implementieren Sie daher in einer Datei `memmanage-mac.h` die folgenden Makros:

```
#define MEM_MAN_ALLOC(ST,P,T,N)   ...
#define MEM_MAN_DELETE_PTR(ST,P)   ...
```

`MEM_MAN_ALLOC` wird mit den Parametern `ST` (Speichertabelle), `P` (Zeigervariable oder `NULL`), `T` (Typ des Zeigers) und `N` (Anzahl der zu allokierenden Elemente) aufgerufen. Das Makro nutzt die beiden Präprozessor-Variablen `__FILE__` und `__LINE__`, um den Namen der Quelldatei und die Zeilennummer des Aufrufs an die Funktion `mem_man_alloc` zu übergeben.

An `MEM_MAN_DELETE_PTR` werden die Parameter `ST` (Speichertabelle) und `P` (Zeiger auf freizugebenden Speicherbereich) übergeben. Dieses Makro ruft entsprechend die Funktion `mem_man_delete_ptr` auf und übergibt ihr auf die oben beschriebene Weise ebenfalls den Namen der Quelldatei und die Zeilennummer des Aufrufs.

In den Materialien zur Übung finden Sie die Datei `memmanage.h` zusammen mit einem Testprogramm `memtest.c`. Dieses benutzt die Funktionen und Makros, die Sie implementieren sollen. Im folgenden sehen Sie die gewünschte Ausgabe des Testprogramms:

- Ausgabe nach `stdout`:

```
Hello World!
1*3*5=15
a[0]=15
a[1]=30
Print out internal memory table:
# active block 0: allocated in file "memtest.c", line 34
# active block 1: allocated in file "memtest.c", line 23
# active block 2: allocated in file "memtest.c", line 24
```

- Ausgabe nach `stderr`:

```
space leak: main memory for block 2 not freed
15 cells of size 1
allocated: file "memtest.c", line 24
```

Die Lösungen zu diesen Aufgaben werden am 23.04.2015 besprochen.