

Schreiben Sie eine Funktion, die jeden ASCII-Code zwischen 32 und 126 zusammen mit dem zugehörigen Zeichen auf dem Bildschirm ausgibt. Über einen Parameter *s* kann bestimmt werden, dass die Ausgabe in *s* Spalten erfolgt. Hier eine Beispielausgabe mit *s* = 10.

```

32   33 !   34 "   35 #   36 $   37 %   38 &   39 '   40 (   41 )
42 *   43 +   44 ,   45 -   46 .   47 /   48 0   49 1   50 2   51 3
52 4   53 5   54 6   55 7   56 8   57 9   58 :   59 ;   60 <   61 =
62 >   63 ?   64 @   65 A   66 B   67 C   68 D   69 E   70 F   71 G
72 H   73 I   74 J   75 K   76 L   77 M   78 N   79 O   80 P   81 Q
82 R   83 S   84 T   85 U   86 V   87 W   88 X   89 Y   90 Z   91 [
92 \   93 ]   94 ^   95 _   96 `   97 a   98 b   99 c  100 d  101 e
102 f  103 g  104 h  105 i  106 j  107 k  108 l  109 m  110 n  111 o
112 p  113 q  114 r  115 s  116 t  117 u  118 v  119 w  120 x  121 y
122 z  123 {  124 |  125 }  126 ~

```

Schreiben Sie eine Funktion, die das Einmaleins auf der Konsole ausgibt. Die Funktion fragt zunächst das kleinste und das größte Einmaleins ab, das berechnet werden soll. Der Anwender muss die Werte über die Tastatur eingeben. Danach erfolgt eine formatierte Ausgabe. Hier ein Beispiel:

Einmaleins

Von:10

Bis:20

```

10  11  12  13  14  15  16  17  18  19  20
20  22  24  26  28  30  32  34  36  38  40
30  33  36  39  42  45  48  51  54  57  60
40  44  48  52  56  60  64  68  72  76  80
50  55  60  65  70  75  80  85  90  95  100
60  66  72  78  84  90  96  102 108 114 120
70  77  84  91  98  105 112 119 126 133 140
80  88  96  104 112 120 128 136 144 152 160
90  99  108 117 126 135 144 153 162 171 180
100 110 120 130 140 150 160 170 180 190 200

```

1. Schreiben Sie eine Funktion `char to_upper_case(char c)`, die einen kleinen Buchstaben in den entsprechenden großen Buchstaben umwandelt (ohne Verwendung der Standardbibliothek).
2. Schreiben Sie eine Funktion `void upper(char text[])`, die alle Buchstaben in einer Zeichenkette in Großbuchstaben umwandelt (ohne Umlaute). Alle Sonderzeichen und Ziffern sollen nicht verändert werden. Verwenden Sie an geeigneter Stelle die Funktion `char to_upper_case(char c)`. Rufen Sie die Funktion `upper` auf und geben Sie das Ergebnis aus. Hinweis: In C gibt es eine enge Verbindung zwischen Zeigern und Feldern. Eine Feldvariable steht für die Speicheradresse, ab der das Feld (Array) beginnt. Bei einem Funktionsaufruf wird damit ein Feld *by reference* übergeben.
3. Implementieren Sie eine *Queue* (Warteschlange) mit einem Array. Die *Queue* soll die beiden Funktionen `enqueue` und `dequeue` besitzen. Mit `void enqueue(int i)` wird die Zahl *i* in die *Queue* aufgenommen. Wenn die *Queue* voll ist, dann wird der Wert am Ende der *Queue* einfach überschrieben. Mit `int dequeue()` wird die Zahl, die am längsten in der *Queue* ist, logisch aus der *Queue* genommen und zurückgeliefert. Wenn die *Queue* leer ist, dann wird -1 zurückgeliefert. Damit der Speicherplatz im Array möglichst gut genutzt wird, realisieren Sie einen logischen Ringspeicher (Sie dürfen die Modulorechnung verwenden). Wenn das Ende des Arrays erreicht ist, dann wird vorne im Array weitergearbeitet, wenn sich dort nicht der Anfang der Warteschlange befindet. Ansonsten wird der Wert am Ende der *Queue* überschrieben. Testen Sie die *Queue* mit dem folgenden Programm. Arbeiten Sie dabei mit einer Arraylänge von 2 für die *Queue*. Die Implementierung soll aber natürlich für beliebige aber feste Arraylängen funktionieren.

```
#include <stdio.h>
#include "queue.h"

main(){
    enqueue(1);
    enqueue(2);
    enqueue(2);
    printf("%i ", dequeue());
    enqueue(3);
    printf("%i ", dequeue());
    printf("%i ", dequeue());
    printf("%i ", dequeue());
    enqueue(4);
    enqueue(5);
    printf("%i ", dequeue());
    printf("%i \n", dequeue());
}
```

Die Ausgabe lautet dann:

1 2 3 -1 4 5

-
1. Schreiben Sie eine Funktion `int countspace(char s[])`, die in einer Zeichenkette die Anzahl aller Leerzeichen ermittelt. Die Funktion gibt die Anzahl der Leerzeichen zurück. Schreiben Sie eine Version mit Indexzugriffen und eine Version mit Zeigerarithmetik.
 2. Schreiben Sie eine Funktion `double min(double a[], int n)`, die den kleinsten Wert im übergebenen Feld ermittelt. Falls das Feld leer ist, soll die Funktion den Wert 0.0 liefern.
 3. Schreiben Sie eine Funktion `char *stringcat (const char* str1, const char* str2)`. Diese Funktion erzeugt eine neue Zeichenkette, indem die Zeichenketten `str1` und `str2` aneinander gehängt werden.
 4. Schreiben Sie eine Funktion, die die Fakultät einer natürlichen Zahl berechnet. Die Fakultät einer natürlichen Zahl n ist wie folgt definiert:

$$n! = \prod_{k=1}^n k \qquad 0! = 1$$

Schreiben Sie eine weitere Funktion, um den Binomialkoeffizienten zweier natürlicher Zahlen zu berechnen. Der Binomialkoeffizient zweier natürlicher Zahlen n und k mit $n \geq k$ ist wie folgt definiert:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Berechnen Sie in einem Hauptprogramm die folgenden Binomialkoeffizienten:

$$\binom{10}{0}, \quad \binom{10}{1}, \quad \binom{10}{10}, \quad \binom{49}{6}$$

Implementieren Sie ein *Dictionary* A unter Verwendung von *Hashing*. Ein *Dictionary* ist eine Menge A , die nur die folgenden Operationen unterstützt:

- **insert**(a): $A \cup \{a\}$
- **delete**(a): $A \setminus \{a\}$
- **member**(a): **true** genau dann, wenn $a \in A$

Das *Dictionary* soll nur ganze Zahlen (**int**) verwalten. Verwenden Sie für die Implementierung das *geschlossene Hashing* (*open addressing*) auf der Basis eines Arrays. Das geschlossene Hashing muss mit m Behältern auskommen. Für eine Eingabe a berechnen Sie mit der Hashfunktion $h(a) = a \bmod m$ eine Behälternummer (den Arrayindex). Für die Operation **insert** speichern Sie dann die Zahl an der berechneten Arrayposition. Sie müssen keine Dubletten verhindern (d.h. eine Zahl darf mehrfach eingetragen werden). Falls eine Kollision auftritt (die berechnete Arrayposition ist bereits belegt), wird unter den übrigen Behältern nach einem freien Platz gesucht. Es gibt verschiedene Verfahren, die sich in der Reihenfolge der besuchten Behälter unterscheiden. Sie verwenden bitte das lineare Sondieren. Beim *linearen Sondieren* (*linear probing*) werden die übrigen Behälter der Reihe nach auf einen freien Platz untersucht. Die Betrachtung eines Behälters wird auch als „Probe“ bezeichnet. Bei der Suche nach einem Schlüssel müssen wir dann die folgenden Fälle unterscheiden:

1. Falls sich der Schlüssel im aufgesuchten Behälter befindet, ist die Suche erfolgreich.
2. Falls der aufgesuchte Behälter leer ist, befindet sich der Schlüssel nicht in der Hash-Tabelle.
3. Falls sich ein anderer Schlüssel in dem Behälter befindet, muss weitergesucht werden. Dabei muss am Ende der Tabelle die Suche am Anfang der Tabelle fortgeführt werden.

Das bedeutet aber auch, dass wir bei der *Delete*-Operation besondere Vorkehrungen treffen müssen. Wir können nicht einfach ein Element löschen, da Elemente, die später eingefügt wurden, auf der Suche nach einem freien Platz das zu löschende Element evtl. übersprungen haben. Da die Suche an einer freien Position stoppt, könnten wir diese Elemente dann später nicht mehr finden. Bitte kennzeichnen Sie die Behälter daher mit einem *Flag*. Dieses *Flag* soll den Status eines Behälters (z.B. frei oder gelöscht) repräsentieren. Die Suche überspringt dann einfach Behälter mit einem gelöschten Inhalt. Diese Behälter können später natürlich wieder gefüllt werden. Für die Darstellung des Status verwenden Sie bitte eine Aufzählung. Für die Arrayelemente definieren Sie bitte eine geeignete Struktur. Verwenden Sie zudem an geeigneten Stellen die Schlüsselwörter **typedef** und **static**. Wenn nicht genügend Speicherplatz zur Verfügung steht, dann liefert die Funktion **insert** den Wert 0, ansonsten den Wert 1. Die Funktion **delete**

gibt den Wert 1 zurück, wenn der Schlüssel *a* gelöscht werden konnte. Ansonsten wird der Wert 0 geliefert.

Für ein Array der Länge 2 sollte die folgende Sequenz z.B. fünfmal eine 1 ausgeben:

```
printf("%d\n", insert(1));  
printf("%d\n", insert(3));  
printf("%d\n", delete(3));  
printf("%d\n", insert(5));  
printf("%d\n", member(5));
```

Implementieren Sie wieder ein *Dictionary A* unter Verwendung von *Hashing* (siehe Praktikum 4). Verwenden Sie für die Implementierung nun aber das *Hashing* mit verketteten Listen (*resolution by chaining*). Dazu definieren Sie ein Array aus m Listenköpfen (diese sollten vom Typ der Listenelemente sein). Für eine Eingabe a berechnen Sie dann mit der Hashfunktion $h(a) = a \bmod m$ eine Behälternummer (den Arrayindex). Für die Operation **insert** nehmen Sie dann die Zahl in die lineare Liste der entsprechenden Arrayposition auf. Sie müssen keine Dubletten verhindern (d.h. eine Zahl darf mehrfach eingetragen werden). Wenn nicht genügend Speicherplatz zur Verfügung steht, dann liefert die Funktion **insert** den Wert 0, ansonsten den Wert 1. Die Funktion **insert** soll in konstanter Laufzeit arbeiten. Für die Operation **member** durchsuchen Sie die Liste an der Arrayposition $h(a)$. Geben Sie nur dann den Wert 1 zurück, wenn sich a in der Liste befindet. Die Funktion **delete** liefert den Wert 1, falls der Schlüssel (das Listenelement) gelöscht werden konnte. Ansonsten wird der Wert 0 geliefert. Als Test lassen Sie die folgende Sequenz ablaufen (SIZE entspricht der Anzahl der Arrayelemente).

```
int i;
for(i = 1; i <= 2 * SIZE; i++){
    printf("%d", insert(i));
}
for(i = 1; i <= SIZE; i++){
    printf("%d", member(i));
}
for(i = SIZE+1; i <= 2*SIZE; i++){
    printf("%d", delete(i));
}
for(i = 1; i <= 2*SIZE; i++){
    printf("%d", member(i));
}
printf("\n");
```

Es muss zunächst eine Folge von $5 * \text{SIZE}$ Einsen ausgegeben werden. Direkt danach erfolgt die Ausgabe von SIZE Nullen.

Schreiben Sie dann ein weiteres Hauptprogramm. Nehmen Sie zunächst alle Zahlen in das Dictionary auf, die über die Kommandozeile übergeben werden. Lesen Sie danach Zahlen über die Tastatur ein. Für jede eingegebene Zahl wird eine Meldung ausgegeben, ob sich die Zahl im Dictionary befindet. Nach Eingabe der Zahl -1 wird das Programm beendet.