# 1 Leistung: Stack vs. Heap

In dieser Aufgabe stehen Ihnen eine Reihe von vorimplementierten Funktionen zur Zeitmessung zur Verfügung. Sie können dazu benutzt werden um relativ präzise Messungen durchzuführen.

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

typedef struct timespec timespec_t;

void capture_time(timespec_t *t1) {
    clock_gettime(CLOCK_MONOTONIC, t1);
}

timespec_t time_diff (const timespec_t end, const timespec_t start) {
    timespec_t diff;
    if (end.tv_nsec < start.tv_nsec) {
        diff.tv_sec = end.tv_sec - start.tv_sec - 1;
        diff.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec;
    } else {
        diff.tv_sec = end.tv_sec - start.tv_sec;
        diff.tv_nsec = end.tv_nsec - start.tv_nsec;
    }
    return diff;
}

double time_to_double (const timespec_t t) {
    double d = (double)t.tv_nsec;
    d /= 1000000000.0;
    d += (double)t.tv_sec;
    return d;
}

int main(int argc, char** args) {
    // your code ...
    return 0;
}
```

Listing 1: Funktionen zur Zeitmessung

Folgender Code zeigt wie eine Messung durchgeführt werden kann.

```c
timespec_t t0, t1;
capture_time(&t0);
// some code
```

```
4  capture_time(&t1);
5  secs = time_to_double(time_diff(t1, t0));
6  printf("duration %0.10f \n", secs);
```

Listing 2: Beispiel einer Zeitmessung

## 1.1 Zeitmessung

Die Messungen sollen 10000 mal wiederholt werden und daraus soll jeweils ein Mittelwert berechnet für die Zeit:

1. die für die Zeitmessung benötigt wird, d.h. wenn zwischen den Messpunkten keine Anweisungen stehen
2. zum Anlegen von gleich grossen Arrays auf dem
    - Stack
    - Heap mit Hilfe von `malloc`
    - Heap mit Hilfe von `calloc`

Das Experiment soll mit Arrays der Grösse von 100, 1000, 10000 und 100000 int-Elementen durchgeführt werden.

## 1.2 Statistiken

Korrigieren Sie Ihre Daten, d.h. ziehen Sie von der mittlerer Allocationszeit, die mittlere Messzeit ab. Die Ausgabe soll folgende Werte beinhalten:

- die mittlere Messzeit.
- den Slow-Down-Faktor, wenn statt auf dem Stack der Speicher mit der `malloc` Funktion auf dem Heap alloziiert wird
- den Slow-Down-Faktor, wenn statt `malloc` der Speicher mit der `calloc` Funktion alloziiert wird
- die Initializierungsgeschwindigkeit, d.h. die Anzahl der Integerwerte pro Sekunde, mit der die `calloc` Funktion initializiert das Array initialisiert hat

Die Ausgabe könnte so aussehen:

```
1 measurement time: 0.023400200000 microseconds
2 malloc slow down factor (compared to stack): 4.683357997099
3 calloc slow down factor (compared to malloc): 522.843097863826
4 initialization rate: 5942387099 ints/sec
```

## 2 Dynamische Speicherallocation

Bei Funktionsaufrufen von `alloc_array` oder `alloc_mem` soll das multidimensionale Array `values` auf dem Heap angelegt werden. Ergänze den folgenden Code.

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

int* alloc_array(size_t dt, size_t dx, size_t dy, size_t dz) {
    // todo: allocate memory
    for(size_t t = 0; t < dt; ++t) {
        for(size_t x = 0; x < dx; ++x) {
            for(size_t y = 0; y < dy; ++y) {
                for(size_t z = 0; z < dz; ++z) {
                    values[t][x][y][z] = t*1000 + x*100 + y*10 + z*1;
                }
            }
        }
    }
    return (int*) values;
}

int* alloc_mem(size_t dt, size_t dx, size_t dy, size_t dz) {
    // todo: allocate memory
    for(size_t t = 0; t < dt; ++t) {
        for(size_t x = 0; x < dx; ++x) {
            for(size_t y = 0; y < dy; ++y) {
                for(size_t z = 0; z < dz; ++z) {
                    size_t idx = dx*dy*dz*t + dy*dz*x + dz*y + z;
                    values[idx] = t*1000 + x*100 + y*10 + z*1;
                }
            }
        }
    }
    return values;
}

int main (int argc, char** argv) {
    // Allocation
    size_t dt = 10, dx = 10, dy = 10, dz = 10;
    int* values_a = alloc_array(dt, dx, dy, dz);
    int* values_p = alloc_mem(dt, dx, dy, dz);

    // Verification
    for (size_t i = 0; i < dt*dx*dy*dz; ++i) {
        printf("%03ld = %04d %04d\n", i, values_a[i], values_p[i]);
    }

    // Cleanup
    // todo: release allocated memory
  return 0;
}
```

# 3 Variable length array

Implementieren Sie ein String-Array, der sich bei Bedarf automatisch erweitert. Die Strategie und die Implementierung ist Ihnen ueberlassen. Der Untere Code dient nur zur Verdeutlichung der Funktionsweise.
Eine gute Strategie könnte sein

- Eine Struktur anzulegen, die
  - `data`: Ein Pointer, der auf den für Daten reservierten Speicher zeigt
  - `capacity`: Die Anzahl der reservierten Elemente speichert
  - `size`: Anzahl der aktuell gespeicherten Elemente darstellt
- Die Struktur zur Beginn mit einer bestimmten Kapazität initialisieren (z.B. 3 Elementen)
- Die Kapazität des Arrays verdoppeln, wenn die Größe `size` an die Kapazität `capacity` stößt

```
1 typedef struct {
2     unsigned int size;
3     unsigned int capacity;
4     char** data;
5 } svector_t;
```

Listing 3: Implementationsvorschlag

Die Modifikaton der Datenstruktur können folgende Operationen übernehmen.

```
1 svector_t* svector_create()
2 void svector_init(svector_t* svector);
3 void svector_append(svector_t*, char* elem);
4 void svector_destroy(svector_t* svector);
5 void svector_print(svector_t* svector);
```

Listing 4: Interface

Hier ist ein Beispiel, wie das Interface benutzt werden kann.

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <stdio.h>
4  #include <stdbool.h>
5
6
7  int main(int argc, char** argv) {
8      svector_t *name_list = svector_create();
9      svector_init(name_list);
10     svector_append(name_list, "Franz Meier");
11     svector_append(name_list, "Tobias Schroeder");
12     svector_append(name_list, "Anne Kraus");
13     svector_append(name_list, "Tom Hook");
14     svector_print(name_list);
```

```
15      svector_destroy(name_list);
16      return 0;
17  }
```