**Fakultät IV - Elektrotechnik und Informatik**
*Einführung in die Programmierung WS 2014/15*
*Feldmann / Rost / Streibelt / Lichtblau / Deimel / Lucas*

# Aufgabenblatt 7

letzte Aktualisierung: 24. Dezember, 14:08 Uhr

(13f65468135155e3d3410282f5f82f1f4138a2e8)

*Ausgabe:* Mittwoch, 17.12.2014
*Abgabe:* spätestens Freitag, 9.01.2015, 18:00

***Thema:*** *Vertiefung des C-Wissens, II*

## Abgabemodalitäten

- Alle abzugebenden Quelltexte müssen ohne Warnungen und Fehler auf den Rechnern des tubIT/IRB mittels `gcc -std=c99 -Wall` kompilieren.
- Abgaben erfolgen prinzipiell immer in Gruppen à 2 Personen, welche in den Tutorien festgelegt wurden. Einzelabgaben sind explizit als solche gekennzeichnet.
- Die Abgabe erfolgt ausschließlich über SVN. Die finale Abgabe
  - für Gruppenabgaben erfolgt im Unterordner `Tutorien/t<xx>/Gruppen/g<xx>/Blatt<xx>/submission/`
  - für Einzelabgaben erfolgt im Unterordner `Tutorien/t<xx>/Studierende/<tuBIT-Login>/Blatt<xx>/submission/`
- Benutzen Sie für alle Abgaben von Programmcode das folgende Namensschema: `introprog_blatt0X_aufgabe0Y_Z.c`, wobei X durch die Blattnummer, Y durch die Aufgabe und Z durch die Unteraufgabe ersetzen ist.
  *Beispiel:* Aufgabe 1.2 wird zu: `introprog_blatt01_aufgabe01_2.c`
  Für jede Unteraufgabe geben Sie maximal eine Quellcodedatei ab, es sei denn, die Aufgabenstellung erfordert explizit die Abgabe mehrerer Dateien pro Aufgabe.

  Alle anderen Abgaben (Pseudocode, Textaufgaben) benennen Sie wie oben beschrieben. Die zugelassenen Abgabeformate sind PDF, ODT und Text (txt). Auch hier verwenden Sie eine Datei pro Aufgabe, nicht jedoch pro Unteraufgabe.



## 1. Aufgabe: Erste Schritte mit Includes (8 Punkte)

In dieser Aufgabe sollen Sie ein kleines Programm schreiben, welches auf eine Datenbank zugreift. Die Datenbank ist in der 'datenbank.c' Datei implementiert. Die Datei 'datenbank.h' enthält die Schnittstelle, mit der Sie auf die Datenbank zugreifen können (siehe Listing 1).

Listing 1: Minimale Schnittstelle zur Datenbank: `datenbank.h`

```
1  //Gibt ein unveränderliches Array der StudentenIds zurück, die bei dem
     ↪ Kurs angemeldet sind.
2  //StudendenIds sind ganze Zahlen größer 0. Das Ende des Arrays wird
     ↪ analog zu Strings durch den Eintrag 0 kenntlich gemacht.
3  const int* lese_Studenten_Ids();
4
5  //Gibt 1 zurück, falls der Student mit der Id studenId die
     ↪ Hausaufgabenvorleistung geschafft hat.
6  //Ansonsten wird 0 zurückgegeben.
7  const int student_hat_Hausaufgaben_bestanden(int studentId);
8
9  //Gibt 1 zurück, falls der Student mit der Id studentId den C-Kurs
     ↪ bestanden hat.
10 //Ansonsten wird 0 zurückgegeben.
11 const int student_hat_CKurs_bestanden(int studentId);
12
13 //Gibt 1 zurück, falls sich der Student mit der Id studentId zur
     ↪ Klausur angemeldet hat.
14 //Ansonsten wird 0 zurückgegben.
15 const int student_hat_Klausuranmeldung(int studentId);
```

**1.1. Benachrichtigung von Studierenden (8 Punkte)** Da die Anmeldung zur Prüfung bis zum 9. Januar 2015 (siehe ISIS) erfolgen muss, sollen Sie herausfinden, für welche der Studierenden alle folgende Eigenschaften zutreffen:

1. Der / die Studierende hat den C-Kurs bestanden.
2. Der / die Studierende hat bereits jetzt 50% der Gesamtpunktzahl der Hausaufgaben erreicht.
3. Der / die Studierende hat sich noch nicht zur Prüfung angemeldet.

Schreiben Sie ein kleines Programm, welches unter Verwendung der in `datenbank.h` spezifizierten Funktionen herausfindet, für welche Studierenden dies zutrifft und geben Sie deren Ids aus. Bei der Beispieldatenbank `datenbank.c` sollte ihr Code nur die Studenten Id `4` ausgeben (wir werden den Code aber natürlich auch mit anderen Beispielen testen).

Zum Einbinden der Datenbank müssen Sie beim Kompilieren
`gcc -std=c99 introprog_blatt07_aufgabe01_1.c datenbank.c`
benutzen und die Datei 'datenbank.h' in Ihrer Abgabe `introprog_blatt07_aufgabe01_1.c` mittels einem Include einbinden. Beachten Sie, dass es unsererseits keine Codevorgabe gibt.

Geben Sie den Quelltext dieser Unteraufgabe in einer Datei mit folgendem Namen ab:
`introprog_blatt07_aufgabe01_1.c`

**Hinweis:** Während die Programmiersprache C z.B. mittels `gcc` auf fast aller Hardware benutzt werden kann, sind die folgenden Aufgaben speziell auf die übliche x86-64 Architektur des herkömmlichen PCs ausgelegt.

## 2. Aufgabe: Dynamische Speicherverwaltung: Blockgrößen und 'invalid writes' (11 Punkte)

In dieser Aufgabe möchten wir maßgeblich anschaulich machen, dass 'buffer overflows', d.h. das Schreiben über Arraygrenzen hinweg, nicht immer sofort zu Fehlern führt. Gerade dies macht diese Art von Fehlern jedoch schwer nachvollziehbar und kostet somit beim Entwickeln von C-Code häufig viel Zeit.

In dem – nicht zur Nachahmung empfohlenen – Code Beispiel 2 wird eine Reihe von Speicherblöcken reserviert sowie deren Adressen und die Abstände zwischen aufeinanderfolgenden Reservierungen ausgewertet. Weiterhin wird am Ende der gesamte Speicher, beginnend bei dem ersten reservierten Speicherblock bis zum Ende des letzten reservierten Speicherblocks, beschrieben.
Wenn Sie das Programm `konsekutives_malloc.c` ausführen, könnten Sie z.B. folgende Ausgabe erhalten:

```
Speicher (  8 Byte): 0xe2c010
Speicher ( 16 Byte): 0xe2c030 (32 Byte von vorigem Block entfernt)
Speicher ( 24 Byte): 0xe2c050 (32 Byte von vorigem Block entfernt)
Speicher ( 32 Byte): 0xe2c070 (32 Byte von vorigem Block entfernt)
Speicher ( 40 Byte): 0xe2c0a0 (48 Byte von vorigem Block entfernt)
Speicher ( 48 Byte): 0xe2c0d0 (48 Byte von vorigem Block entfernt)
Speicher ( 56 Byte): 0xe2c110 (64 Byte von vorigem Block entfernt)
...
```

Die folgenden Fragen sind sehr allgemein gehalten, da sie maßgeblich zum Nachdenken anregen sollen. Bitte versuchen Sie sich bei der Beantwortung möglichst kurz zu fassen. Bei den meisten Aufgaben reichen wenige Sätze.

**2.1. Beobachtung (1 Punkte)** Vergleichen Sie die Abstände zwischen den reservierten Speicherblöcken mit der Größe des reservierten Speichers an Hand der obigen Beispielausgabe. Was fällt Ihnen auf?

**2.2. Beobachtung (1 Punkte)** In der obigen Beispielausgabe liegen die reservierten Speicherbereiche immer hintereinander. Führen Sie den Programmcode auf Ihrem Rechner bzw. einem IRB-Rechner aus, welche Ausgabe erhalten Sie und inwieweit deckt sich diese mit obiger Ausgabe? (Bitte geben Sie Ihre Ausgabe ab.)

**2.3. Beobachtung (1 Punkte)** Der obige Programmcode stürzt – zumindest auf dem Testrechner – nicht ab. Welche Ausgabe erhalten Sie, wenn Sie obiges Programm mit `valgrind -v` ausführen?

**2.4. Beobachtung (2 Punkte)** Ändern Sie das Programm ab indem Sie z.B. weiteren Speicher beliebig allozieren und gleich wieder befreien. Sie können z.B. in der `for`-Schleife das Kommando `free(malloc(40*(i+1)* sizeof(char)));` benutzen. Inwieweit stimmen Sie der folgenden Aussage zu: "Mehrere Aufrufe von `malloc` liefern immer aufsteigend bzw. absteigend sortierte Speicherbereiche zurück. Der Heap wächst immer in eine Richtung."? (Bitte geben Sie Ihre Ausgabe ab.)

**2.5. Speicherfreigabe (3 Punkte)** Auch wenn obiger Programmcode natürlich wenig Sinn macht, sollte der Speicher immer freigegeben werden. Erweitern Sie obiges Programm, so dass direkt vor `return 0;` alle reservierten Speicherbereiche mittel `free` freigegeben werden. Was passiert, wenn Sie dieses Programm ausführen? Versuchen Sie das Verhalten zu erklären.

**2.6. Schlussfolgerung (3 Punkte)** Betrachten Sie folgendes Programmfragment:



Listing 3: `speicherverletzung_mini_beispiel.c`

```
1 int i = 12;
2 char* string = malloc(i);
3 string[i+1] = ~string[i+1];
4 free(string);
```

1. Stellen Sie fest, ob und für welche ganzzahligen `i` obiger Code einen Laufzeitfehler erzeugt. Schreiben Sie dazu ein minimales Programm, welches den Code in einer for-Schleife für das obige `i` von 1 bis 64 testet (dieses Programm muss nicht abgegeben werden).
2. Erklären Sie beispielhaft warum man – z.B. mittels `valgrind` – immer sicherstellen sollte, dass man nur korrekte Speicherzugriffe durchführt. Beziehen Sie dabei insbesondere auch ihre Beobachtungen 2.1. und 2.4. ein.

Geben Sie Ihre Lösung der Aufgabe in einem Textdokument mit einem der folgenden Namen ab:
`introprog_blatt07_aufgabe02.{txt|odt|pdf}`

Code Beispiel 2: Allokation von Speicherblöcken, Vergleich der Abstände der Blöcke mit den reservierten Größen sowie Überschreiben des gesammelten Speicherbereichs mit einem Wert (konsekutives_malloc.c).

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

#define LAENGE 16

//schreibt das 'zeichen' in alle Speicherzellen zwischen start und ende
void schreibeZeichen(char* start, char* end, char zeichen)
{
   if(start > end)
   {
      char* tmp = start;
      start = end;
      end = tmp;
   }
   while(start <= end)
   {
      *start = zeichen;
      start++;
   }
}

int main()
{
   char* stringArray[LAENGE];
   for(int i = 0; i < LAENGE; ++i)
   {
      int blockGroesse = (i+1)*8;
      stringArray[i] = malloc(blockGroesse * sizeof(char));
      strcpy(stringArray[i], "...");
      if(i == 0)
      {
         printf("Speicher (%3d Byte): %p\n", blockGroesse, stringArray[i]);
      }
      else
      {
         printf("Speicher (%3d Byte): %p (%ld Byte von vorigem Block entfernt )\n", blockGroesse, stringArray[i], stringArray[i]-stringArray[i-1]);
      }
   }

   char* start = stringArray[0];
   char* ende = stringArray[LAENGE-1] + LAENGE*8;

   schreibeZeichen(start, ende, '0');

   return 0;
}
```



## 3. Aufgabe: Stacks (6 Punkte)

**Hinweise:** Zur Lösung dieser Aufgabe können die Folien der Vorlesung vom 17.12.2014 sehr hilfreich sein. Weiterhin reicht auch bei dieser Aufgabe eine kurze (aber präzise) Begründung aus.

Betrachten Sie den Code in Beispiel 4 und führen Sie ihn auf Ihrem Rechner aus. Erklären Sie warum wir Euch niemals 'böse Weihnachten' wünschen könnten bzw. präziser: Warum wird dieser Code auf einer regulären x86-Architektur immer – noch bevor das unsägliche `printf` in der Funktion `main` erreicht wird – niemals wieder zurück in die main() Methode finden können?
Geben Sie Ihre Lösung der Aufgabe in einem Textdokument mit einem der folgenden Namen ab:
`introprog_blatt07_aufgabe03.{txt|odt|pdf}`

Code Beispiel 4: Gefährlich? (stack_smashing.c).

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

#define LAENGE 20

void schreibeZeichen(char* start, char* end, char zeichen)
{
   if(start > end)
   {
      char* tmp = start;
      start = end;
      end = tmp;
   }
   printf("beschreibe Speicher von %p bis %p mit %c..\n", start, end, zeichen );
   while(start <= end)
   {
      *(start++) = zeichen;
      start++;
   }
}

void smash_it(char* speicher_aufrufer){
   char speicher_hier[LAENGE];
   schreibeZeichen(speicher_aufrufer, speicher_hier, '\0');
   printf("Das 'Einführung in die Prorgammierung'-Team wünscht schöne Feiertage und einen guten Rutsch ins neue Jahr!\n");
}


int main(){
   char speicher_hier[LAENGE];
   printf("Nur die letzt genannte Aussage zählt!\n");
   smash_it(speicher_hier);
   printf("Das 'Einführung in die Prorgammierung'-Team wünscht schreckliche Feiertage und einen schlechten Rutsch ins neue Jahr!\n");
   return 0;
}
```