RA FS 18 Lösung Repetitionsserie: Teil C

Adrian Wachili, Givi Meishvili, Mauro Kiener, Qiyang Hu

Die Repetitionsserie dient der individuellen Prüfungsvorbereitung und muss nicht abgegeben werden. Für Fragen steht im ILIAS jederzeit ein Forum zur Verfügung.
Viel Spass!

1 C-Programmierung

1.1 Pointer (1 Punkt)

Angenommen `x` werde im untenstehenden Programm an der Adresse `0xbffff97c` initialisiert (`0x` bedeutet, dass die Adresse `bffff97c` hexadezimal ist).

Auf welche Adresse im Speicher zeigt `px` nach Ausführung des Programms? Gehen Sie von einem 32-Bit Prozessor aus.

```
1 main() {
2   short x, *px;
3   px = &x;
4   px++;
5 }
```

1.2 StringCopy (1 Punkt)

Der folgende Code-Ausschnitt zeigt eine typische Implementation der `strcpy`-Funktion. Welche Probleme können auftreten?

```
1 char *strcpy(char *s1, const char *s2) {
2   char *dst = s1;
3   const char *src = s2;
4   while( (*dst++=*src++)!='\0');
5   return s1;
6 }
```

1.3 Sizeof

Welche Ausgabe erzeugt das folgende Codestück?

```
1 char a[10];
2 char* b = a;
3 printf("%i %i\n",sizeof(a), sizeof(b));
```

Lösung:

Die Ausgabe lautet `10 4` (auf einem 32-Bit System), das Array besteht aus zehn Chars, die je ein Byte Platz benötigen, während der Pointer eine 32-Bit Adresse speichert, also 4 Byte.

1.4 Arrays initialisieren

Wieso kann in C eine Array-initialisierende Funktion nicht folgendes machen?

```
1 int* init() {
2   int i[32];
3   return i;
4 }
```

Lösung:

Das Array wird nur auf dem lokalen Stack der Funtion `init` erzeug, es wird zwar ein Pointer auf die entsprechende Speicherstelle zurückgegeben, aber nach dem Beenden der Funktion wird dieser Speicher wieder freigegeben.

1.5 Pointerarithmetik

Wies gibt `int *a = address; a++;` und `char *a = address; a++;` nicht das selbe Resultat?.
Lösung:

Durch die Pointerarithmetik wird beim `++` jeweils `+sizeof(a)` gerechnet, im ersten Fall also (normalerweise) `+4`, im zweiten Fall `+1`.

1.6 Variablentypen

Welchen Typ haben die Variablen aus `int* a, b`? **Lösung:**

`a` ist ein Pointer auf einen Integer, `b` ist ein Integer.

Will man zwei Pointer, so muss dies mittels `int *a, *b` deklariert werden. Dies zeigt auch, wieso es von Vorteil ist, den `*` jeweils bei der Variable zu notieren.

1.7 Operatorenpräzedenz

In welcher Reihenfolge werden die Ausdrücke in `*&a++` evaluiert?

Lösung:

Postfix `++` hat die höchste Präzedenz, `*` und `&` haben gleiche Präzedenz und werden von rechts nach links evaluiert. Die Reihenfolge sieht also wie folgt aus

Zuerst wird `++` ausgeführt, d.h. `a` selber wird um Eins erhöht, die nachfolgenden Operatoren arbeiten aber noch mit dem alten Wert von `a`.

Anschliessend wird die Adresse von `a` mittels `&` bestimmt und wiederum mittels `*` dereferenziert.

1.8 Best coding practices

Warum ist `if (3 == a) {}` besser als `if (a == 3) {}`?

Lösung:

Schreibt man aus Versehen `if (a = 3)`, dann wird zum einen `a` der Wert 3 zugewiesen, zum anderen wird die `if`-Bedingung immer war sein, da der Rückgabewert von `a = 3` der Wert 3 ist. Ein solcher Fehler wird beim Durchsehen des Codes sehr schwer zu entdecken sein.

Schreibt man hingegen versehentlich `if (3 = a)`, dann wird dies zu einem Fehler beim Compilieren führen, da 3 kein sogenannter L-Value ist, d.h. ein Wert, dem man etwas zuweisen kann (wie z.B. eine Variable).

1.9 Preprocessor Fun

Welche Ausgabe erzeugt das folgende Codestück?

```
1 #define MIN(a,b) ((a) < (b) ? (a) : (b))
2
3 int a=3;
4 int b=4;
5 int m=MIN(a++,b++);
6 printf("min(%i, %i) = %i\n", a, b, m);
```

Lösung:

Die Ausgabe lautet `min(5, 5) = 4`.

`m=MIN(a++,b++);` wird ersetzt durch `m=((a++) < (b++) ? (a++) : (b++))`. Es wird also zuerst überprüft, ob `a<b` und anschliessend werden `a` und `b` auf 4 bzw. 5 erhöht (Post-Inkrement). Da `a<b`, findet die Zuweisung `m=a++` statt, d.h. `m` erhält den Wert 4 und `a` wird noch einmal erhöht auf 5.

1.10 Datenstrukturen in C

Um was für eine Struktur handelt es sich bei `tr`? Stellen Sie `tr* a` graphisch dar und beschreiben Sie Output und Ablauf des folgenden Programms

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  typedef struct tr_ {
5    char n;
6    struct tr_ *l;
7    struct tr_ *r;
8  } tr;
9
10 tr* ct(char n, tr *l, tr *r) {
11    tr *t = malloc(sizeof(t));
12    t->n = n; t->l = l; t->r = r;
13    return t;
14 }
15
16 void rec(tr *t, void (*o)(tr *)) {
17    if (t) {
18       tr *l = t->l;
19       tr *r = t->r;
20       o(t);
21       rec(l, o);
22       rec(r, o);
23    }
24 }
25
26 void p(tr *t) {
27   printf("%c", t->n);
28 }
29
30 void s(tr *t) {
31   tr* tmp = t->l;
32   t->l = t->r;
33   t->r = tmp;
34 }
35
36 void fr(tr *t) {
37   free(t);
38 }
39
40 int main() {
41   tr* g = ct('g',NULL,NULL);
42   tr* f = ct('f',NULL,NULL);
43   tr* e = ct('e',NULL,NULL);
44   tr* d = ct('d',NULL,NULL);
45   tr* c = ct('c',f,g);
46   tr* b = ct('b',d,e);
47   tr* a = ct('a',b,c);
48   rec(a, &p);
49   printf("\n");
50   rec(a, &s);
51   rec(a, &p);
```

```
52   printf("\n");
53   rec(a, &fr);
54   return EXIT_SUCCESS;
55 }
```

Lösung:

Die Struktur `tr` ist ein binärer Baum (tree).

Der Output des Programms lautet

```
abdecfg
acgfbed
```

Die Funktion `ct` (create tree) alloziert Speicher für einen Knoten des Baumes, erstellt einen Knoten mit den entsprechenden Kindern und gibt einen Pointer auf den neuen Knoten zurück.

Die Funktion `rec` führt eine Tiefensuche (depth-first) auf dem Baum durch und wendet dabei die Funktion `o` auf jeden Knoten an. Folgende Operationen werden ausgeführt:

(a) Baum ausdrucken (`p` d.h. print)

(b) Knoten im Baum vertauschen (links wird rechts und umgekehrt) (`s` d.h. swap)

(c) Baum noch einmal ausdrucken

(d) Knoten des Baums freigeben (`fr` d.h. free)

1.11 Mehr Fehlersuche

Das folgende Programm gibt den Output `(null) (null), 0`. Wo liegt der Fehler? Korrigieren Sie die Fehler, so dass das Programm die intendierte Funktion erbringt.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define MAX_STRING_LENGTH 40
5
6  typedef struct {
7    char *firstName;
8    char *lastName;
9    unsigned int age;
10 } Person;
11
12 void assignValues(Person p, const char *firstName, const char *lastName, unsigned int age) {
13   p.firstName = firstName;
14   p.lastName = lastName;
15   p.age = age;
16 }
17
18 int main(int argc, char** argv) {
19   Person p = *(Person*)malloc(sizeof(Person));
20   assignValues(p, "Sandra", "Muster", 33);
21   printf("%s %s, %i\n", p.firstName, p.lastName, p.age);
22   return EXIT_SUCCESS;
23 }
```

Lösung:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define MAX_STRING_LENGTH 40
5
6  typedef struct {
```

```
 7   const char *firstName;
 8   const char *lastName;
 9   unsigned int age;
10 } Person;
11
12 void assignValues(Person *p, const char *firstName, const char *lastName, unsigned int age)
13   (*p).firstName = firstName;
14   (*p).lastName = lastName;
15   (*p).age = age;
16 }
17
18 int main(int argc, char** argv) {
19   Person *p = *(Person*)malloc(sizeof(Person));
20   assignValues(p, "Sandra", "Muster", 33);
21   printf("%s %s, %i\n", (*p).firstName, (*p).lastName, (*p).age);
22   return EXIT_SUCCESS;
23 }
```