Programmierung 2 (SS 2021)
Universität des Saarlandes
Fakultät MI
Compiler Design Lab

Musterlösung 5
C

Prof. Dr. Sebastian Hack
Julian Rosemann, B. Sc.
Thorsten Klößner, M. Sc.
Julia Wichlacz, M. Sc.
Maximilian Fickert, M. Sc.

Im Vorlesungskalender finden Sie Informationen über die Kapitel des Skripts, die parallel zur Vorlesung bearbeitet werden sollen bzw. dort besprochen werden. Die Übungsaufgaben dienen der Vertiefung des Wissens, das in der Vorlesung vermittelt wird und als Vorbereitung auf Minitests und Klausur.
Weitere Aufgaben zu den Themen finden Sie jeweils am Ende der Skriptkapitel.
Die Schwierigkeitsgrade sind durch Steine des 2048-Spiels gekennzeichnet, von 512 „leicht“ bis 2048 „schwer“. 4096 steht für Knobelaufgaben.

Hinweis: Die ersten 3 Aufgaben finden Sie bereits auf Blatt 4. Damit Sie eine bessere Übersicht über die C-Aufgaben haben, haben wir diese trotzdem auch diesem Blatt hinzugefügt.

Aufgabe 5.0: Kompilieren und ausführen

1. Schreiben Sie ein C-Programm, welches zwei Variablen vom Typ `int` anlegt, ihnen Werte zuweist und die Werte der Variablen auf der Konsole ausgibt. Erstellen Sie dazu eine Datei `main.c` und schreiben Sie die entsprechende `main`-Funktion.
   Erstellen Sie eine ausführbare Datei `prog` und führen Sie diese aus.

2. Erweitern Sie die Datei `main.c` aus Aufgabenteil (a) um eine Funktion `max`, welche zwei Argumente vom Typ `int` bekommt und das Maximum der beiden Werte zurückgibt.
   Erstellen Sie eine ausführbare Datei `prog` und führen Sie diese aus.

3. Lagern Sie die in Aufgabenteil (b) erstellte Funktion `max` in eine separate Übersetzungseinheit `max.c` aus. Verwenden Sie eine Headerdatei `max.h`, um Ihrem Hauptprogramm den Prototypen des Unterprogramms `max` bekannt zu machen.
   Erstellen Sie eine ausführbare Datei `prog` und führen Sie diese aus.

Lösung

1. Programm

```c
#include <stdio.h>

int main() {
  int a = 10;
  int b = 20;

  printf("a: %d\n", a);
  printf("b: %d\n", b);

  return 0;
}
```

   Ausführbares Programm `prog` erstellen und ausführen.

```
$ cc -o main.o -c main.c
$ cc -o prog main.o
$ ./prog
```

   Kurz:

```
$ cc main.c -o prog
$ ./prog
```

2. Funktion `max`

```c
#include <stdio.h>

int max(int x, int y) {
  if(x > y) {
    return x;
  } else {
    return y;
  }
}

int main() {
  int a = 10;
  int b = 20;

  int res = max(a, b);

  printf("Maximum von a und b: %d\n", res);

  return 0;
}
```

Ausführbares Programm `prog` erstellen: Wie oben.

3. Mit header-Datei
   max.h:

```c
#ifndef MAX_H
#define MAX_H

int max(int x, int y);

#endif
```

max.c:

```c
#include "max.h"

int max(int x, int y) {
  if(x > y) {
    return x;
  } else {
    return y;
  }
}
```

main.c:

```c
#include <stdio.h>
#include "max.h"

int main() {
  int a = 10;
  int b = 20;

  int res = max(a, b);

  printf("Maximum von a und b: %d\n", res);

  return 0;
}
```

Ausführbares Programm `prog` erstellen und ausführen:

```
$ cc -o main.o -c main.c
$ cc -o max.o -c max.c
$ cc -o prog main.o max.o
$ ./prog
```

Kurz:

```
$ cc main.c max.c -o prog
$ ./prog
```

Aufgabe 5.1: C Programme verstehen

Untersuchen und verstehen Sie das folgende Programm:

```
1  #include <stdio.h>
2  int main() {
3      for (int i = 1; i < 10; i++) {
4          if ((i % 2) == 0) {
5              printf("Die␣Zahl␣%i␣ist␣...\n", i);
6          } else {
7              printf("Die␣Zahl␣%i␣ist␣...\n", i);
8          }
9      }
10     return 0;
11 }
```

1. Welche Ausgabe liefert es? Wie können Sie die Pünktchen in der Ausgabe sinnvoll ersetzen?
2. Erweitern Sie das Programm so, dass die Ausgabe für den Bereich von $0 - 20$ (inklusive Grenzen) erfolgt.
3. Können Sie die Schleife so abändern, dass die Zahlen aus Teil (b) rückwärts ausgegeben werden?

Lösung

1. Die ersten Pünktchen sind sinnvoller durch gerade und die zweiten Pünktchen durch ungerade zu ersetzen. Dann liefert das Programm folgende Ausgabe:

```
Die Zahl 1 ist ungerade
Die Zahl 2 ist gerade
Die Zahl 3 ist ungerade
Die Zahl 4 ist gerade
Die Zahl 5 ist ungerade
Die Zahl 6 ist gerade
Die Zahl 7 ist ungerade
Die Zahl 8 ist gerade
Die Zahl 9 ist ungerade
```

2. Erweiterung

```
1  #include <stdio.h>
2  int main() {
3    for (int i = 0; i <= 20; i++) {
4      if ((i % 2) == 0) {
5        printf("Die␣Zahl␣%i␣ist␣gerade\n", i);
6      } else {
7        printf("Die␣Zahl␣%i␣ist␣ungerade\n", i);
8      }
9    }
10   return 0;
11 }
```

3. Rückwärts

```c
#include <stdio.h>
int main() {
  for (int i = 20; i >= 0; i--) {
    if ((i % 2) == 0) {
      printf("Die Zahl %i ist gerade\n", i);
    } else {
      printf("Die Zahl %i ist ungerade\n", i);
    }
  }
  return 0;
}
```

Aufgabe 5.2: C Programme verstehen und erweitern

Konrad Klug hat das Kapitel zu C im Skript gelesen und fühlt sich nun bereit, sein erstes C-Programm zu schreiben. No Hau hat ihm erzählt, dass ihr erstes Programm die Zahlen von 0 bis 100 ausgegeben hat, was Konrad jedoch nicht reicht. Er will nun die Zahlen von 0 bis 100 in umgekehrter Reihenfolge und nur in 5er Schritten ausgeben. Also: 100, 95, 90, ..., 5, 0.

1. Konrad ist mit dem Programm fast fertig. Ihm fehlt nur noch ein wesentlicher Baustein. Ersetzen Sie im folgenden Programm den Kommentar `/* hier fehlt was */` so, dass es die oben genannte Zahlenfolge ausgibt:

```c
int main() {
  int j = 100;
  int z = 5;
  do {
    /* hier fehlt was */
  } while (j >= 0);
  return 0;
}
```

2. Was passiert, wenn die Variable j einen Wert kleiner Null zugewiesen bekommt? Würde sich das ändern, wenn wir anstelle einer offenen Schleife eine geschlossene Schleife verwenden?

3. Nachdem Konrad seine Zahlenfolge nun sehen konnte würde er dasselbe auch gerne mit anderen Zahlen ausprobieren. Er hat aber von No Hau gehört, dass es schlecht sei Code zu duplizieren. Um das zu vermeiden, hat er sich vorgenommen ein kleines Unterprogramm zu schreiben. Dieses soll abhängig von beliebig gewählten Zahlen j und z die Zahlenfolge $j, j-z, j-2z, ..., j-\lfloor \frac{j}{z} \rfloor z$ ausgibt. Leider erinnert er sich nicht mehr genau wie man das macht. Vervollständigen Sie seinen Code analog zu Aufgabenteil (a), jedoch unter Verwendung einer geschlossenen Schleife.

```c
int main() {
  magic(100, 5);
  magic(100, 10);
  magic(5000, 200);
  return 0;
}

void magic(...) {
  ...
}
```

Lösung

1. vervollständigtes Programm:

```
1  int main() {
2    int j = 100;
3    int z = 5;
4    do {
5      printf("Die␣Zahl␣ist:␣%i␣\n", j ) ;
6      j -= z;
7    } while (j >= 0);
8    return 0;
9  }
```

2. Wenn die Variable j einen Wert kleiner Null zugewiesen bekommt, wird die Schleife einmal ausgeführt, obwohl die Abbruchbedingung gilt. Im Fall einer geschlossenen Schleife würde der Schleifenrumpf überhaupt nicht ausgeführt werden.

3. modulare Version:

```
1  void magic(int j, int z){
2    while (j >= 0) {
3      printf("Die␣Zahl␣ist:␣%i␣\n", j ) ;
4      j-=z;
5    }
6  }
7
8  int main(){
9    magic(100, 5);
10   magic(100, 10);
11   magic(5000, 200);
12   return 0;
13 }
```

Aufgabe 5.3: Schaltjahr

Nach dem gregorianischen Kalender ist jedes Jahr ein Schaltjahr, das die 3 folgenden Bedingungen erfüllt:

1. Die Jahreszahl ist ganzzahlig durch 4 teilbar.
2. Die Jahreszahl ist nicht ganzzahlig durch 100 teilbar.
3. Ist sie ganzzahlig durch 400 teilbar entfällt Regel 2.

Schreiben Sie ein C-Unterprogramm `int schalt(int jahr)` welches für eine Eingabe entscheidet, ob es sich um ein Schaltjahr handelt. Falls ja, soll Ihr Unterprogramm den Wert 1 zurückgeben, andernfalls den Wert 0.

Lösung

```
1 int schalt(int jahr) {
2   // 4 teilt jahr nicht -> false
3   if (jahr%4 != 0){
4     return 0;
5   }
6   // 4 teilt jahr
7   // 100 teilt jahr
8   if (jahr%100 == 0){
9     // 400 teilt jahr -> true
```

```
10      if (jahr%400 == 0){
11        return 1;
12      // 400 teilt jahr nicht -> false
13      } else {
14        return 0;
15      }
16    }
17    // 100 teilt jahr nicht -> true
18    return 1;
19  }
```

Aufgabe 5.4: Zweierpotenzen

1. Konrad Klug liebt es mit Binärzahlen zu rechnen. Allerdings hat er Probleme damit sich die ganzen Zweierpotenzen zu merken. Helfen Sie Konrad, indem Sie ein C-Unterprogramm `int pow2(int n)` schreiben, das 2^n berechnet.

2. Erweitern Sie Ihr Programm so, dass Konrad nicht nur Zweierpotenzen, sondern auch eine beliebige Base a mit n potenzieren kann, sodass a^n berechnet wird.

Lösung

```
1   //zu 1
2   int pow(int exp, int base);
3
4   int pow2(int exp){
5     return pow(exp, 2);
6   }
7
8   //zu 2
9   int pow(int exp, int base) {
10    int i = 0;
11    int erg = base;
12    while(i < exp) {
13      erg *= base;
14      i++;
15    }
16    return erg;
17  }
```

Aufgabe 5.5: Zeiger überall

1. Konrad Klug und No Hau haben heute gelernt, was Pointer in C sind. Deswegen haben sie gleich losgelegt und ihre ersten Programme geschrieben, in denen sie mit Pointern experimentiert haben. Leider haben sich in Konrads ersten Zuweisungsblock Syntaxfehler eingeschlichen. Ermitteln Sie welcher der inkorrekte Zuweisungsblock ist und helfen Sie Konrad, indem Sie die fehlerhaften Stellen markieren.

```
1   int x = 20;
2   int y = 11;
3   int* px = &x;
4   int *py = &y;
```

```
1   int x = 20;
2   int y = 11;
3   *int px = &x;
4   int py* = &y;
```

2. Nachdem Konrad mit Ihrer Hilfe seine Fehler gefunden hat, arbeitet er weiter an seinem Programm. Betrachten Sie die folgende Tabelle mit Konrads Programm und vervollständigen Sie diese. Entscheiden Sie zuerst

für jede Zuweisung, ob sie gültig ist oder nicht. Falls die Zuweisung gültig ist, geben Sie an in welchem Zustand sich die Variablen nach der Zuweisung befinden. Gehen Sie dabei zu Beginn der Aufgabe von dem gültigen Zuweisungsblock aus Aufgabenteil (a) aus und zu Beginn einer jeden Zeile vom Zustand der letzten gültigen.

	Zuweisung	**x**	**y**	**px**	**py**
0.	-	20	11	0x004	0x008
1.	`*px = 7;`	7	11	0x004	0x008
2.	`&px = &py;`	ungültig			
3.	`px = &y;`				
4.	`&px = *x;`				
5.	`*px = 5;`				
6.	`*x = *py;`				
7.	`y = y + x;`				
8.	`*x = *y;`				
9.	`x = (px == py);`				
10.	`&x = py;`				
11.	`x = x + y;`				
12.	`px = &x;`				
13.	`x = *(px + 4);`				

3. Weil No Hau gemerkt hat, dass Konrad noch mehr Schwierigkeiten mit Pointern hat als Sie, hat er das folgende Quiz entworfen. Konrad glaubt, dass er alle Fragen korrekt beantworten kann. Können Sie das auch?

Frage 1: Welchen Typen haben die Variablen? Sind alle vier Deklarationen gültig?

(a) `int* a, b;`

(b) `int *a, b;`

(c) `int *a, *b;`

(d)

```
1 int  x = 11;
2 int  *px = &x;
3 int* *a = &px;
```

Frage 2: Welche Ausgaben erzeugen die folgenden Ausdrücke? Nehmen Sie an, dass `px` und `a` analog zu 4. aus Frage 1 definiert wurden.

(a) `printf("%d", *px);`

(b) `printf("%d", *a);`

Frage 3: Sind die folgenden beiden Ausdrücke gültig? Falls ja, zu was werten sie aus? Nehmen Sie erneut an, dass `px` und `a` analog zu 4. aus Frage 1 definiert wurden.

(a) `x = (&px == (a + 4));`

(b) `x = (&(px + 4) == (a + 4));`

Lösung

1. Der 1. Block ist gültig! Im zweiten Block sind die beiden letzten Zeilen keine gültigen Zuweisungen.

2.

	Zuweisung	x	y	px	py
0.	-	20	11	0x004	0x008
1.	`*px = 7;`	7	11	0x004	0x008
2.	`&px = &py;`	ungültig			
3.	`px = &y;`	7	11	0x008	0x008
4.	`&px = *x;`	ungültig			
5.	`*px = 5;`	7	5	0x008	0x008
6.	`*x = *py;`	ungültig			
7.	`y = y + x;`	7	12	0x008	0x008
8.	`*x = *y;`	ungültig			
9.	`x = (px == py);`	1	12	0x008	0x008
10.	`&x = py;`	ungültig			
11.	`x = x + y;`	13	12	0x008	0x008
12.	`px = &x;`	13	12	0x004	0x008
13.	`x = *(px + 4);`	undefiniertes Verhalten			

3. **Frage 1:** Alle vier Deklarationen sind gültig.

   1. `a: int*, b: int`
   2. `a: int*, b: int`
   3. `a: int*, b: int*`
   4. `x: int , px: int* , a: int**`

   Frage 2:

   (a) `printf("%d", *px);` gibt 11 aus

   (b) `printf("%d", *a);` gibt die Adresse aus, an der 11 gespeichert ist

Frage 3:

(a) `x = (&px == (a + 4));` gültig, x wertet zu 0 aus, da `a == &px` ist und somit `(a+4) != &px`.

(b) `x = (&(px + 4) == (a + 4));` kein gültiger Ausdruck, da `px+4` kein L-Value ist

Aufgabe 5.6: Eine Partie Pointergolf

Versuchen Sie jede einzelne Anweisung nachzuvollziehen. Erzeugt Ihr Programm auch die Ausgabe „56-42-13“?

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  int x = 56;
  int *ap, *bp;
  int **app, **bpp;

  int r[3];

  r[0] = 1;
  r[1] = 2;
  r[2] = 3;

  ap = &r[1];
  bp = &r[2];

  app = &ap;
  bpp = &bp;

  *ap = 43;
  *(*(bpp)) = 13;
  ap++;
  bp -= 2;
  *(bp + 1) = 17;
  ap = &x;
  app = &bp;
  *bp = *ap;
  bpp = &ap;
  ap = &r[2];
  *((*bpp) - 1) = 42;

  printf("%d-%d-%d\n", r[0], r[1], r[2]);
}
```

Lösung

Das Programm liefert die Ausgabe:

```
56-42-13
```

Nr.	x	ap	bp	app	bpp	r[0]	r[1]	r[2]
8	56	?	?	?	?	?	?	?
10	56	?	?	?	?	1	?	?
11	56	?	?	?	?	1	2	?
12	56	?	?	?	?	1	2	3
14	56	&r[1]	?	?	?	1	2	3
15	56	&r[1]	&r[2]	?	?	1	2	3
17	56	&r[1]	&r[2]	&ap	?	1	2	3
18	56	&r[1]	&r[2]	&ap	&bp	1	2	3
20	56	&r[1]	&r[2]	&ap	&bp	1	43	3
21	56	&r[1]	&r[2]	&ap	&bp	1	43	13
22	56	&r[2]	&r[2]	&ap	&bp	1	43	13
23	56	&r[2]	&r[0]	&ap	&bp	1	43	13
24	56	&r[2]	&r[0]	&ap	&bp	1	17	13
25	56	&x	&r[0]	&ap	&bp	1	17	13
26	56	&x	&r[0]	&bp	&bp	1	17	13
27	56	&x	&r[0]	&bp	&bp	56	17	13
28	56	&x	&r[0]	&bp	&ap	56	17	13
29	56	&r[2]	&r[0]	&bp	&ap	56	17	13
30	56	&r[2]	&r[0]	&bp	&ap	56	42	13

Nr. gibt dabei die Zeilennummer an, wobei in der Tabelle der Zustand unmittelbar nach der Ausführung der entsprechenden Zeile vermerkt ist.

Aufgabe 5.7: Pointerverfolgung kann man nie genug üben

Bestimmen Sie von Hand, welche Ausgabe das folgende Programm liefert. Benutzen Sie den Computer nur, um zu überprüfen, ob Ihr Ergebnis korrekt ist.

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]) {
4     int aa[6] = { 50, 60, 70, 80, 90, 100 };
5     int bb[6] = { -2, 2, 0, 1, -1, 3 };
6
7     int *a, **b, *c, **d, *e;
8
9     c = &bb[5];
10    d = &c;
11    a = bb;
12    b = &a;
13
14    for (e = *b; e <= *d; e++) {
15        *e = *(aa + 3 - *e);
16    }
17
18    printf("%d␣%d␣%d␣%d␣%d␣%d\n", bb[0], bb[1], bb[2], bb[3], bb[4], bb[5]);
19
20    return 0;
21 }
```

Lösung

Ausgabe des Programms: 100 60 80 70 90 50

Schritt	Zeile	a	b	c	d	e	aa	bb
1	4	-	-	-	-	-	[50, 60, 70, 80, 90, 100]	-
2	5	-	-	-	-	-	[50, 60, 70, 80, 90, 100]	[-2, 2, 0, 1, -1, 3]
3	7	?	?	?	?	?	[50, 60, 70, 80, 90, 100]	[-2, 2, 0, 1, -1, 3]
4	9	?	?	&bb[5]	?	?	[50, 60, 70, 80, 90, 100]	[-2, 2, 0, 1, -1, 3]
5	10	?	?	&bb[5]	&c	?	[50, 60, 70, 80, 90, 100]	[-2, 2, 0, 1, -1, 3]
6	11	&bb[0]	?	&bb[5]	&c	?	[50, 60, 70, 80, 90, 100]	[-2, 2, 0, 1, -1, 3]
7	12	&bb[0]	&a	&bb[5]	&c	?	[50, 60, 70, 80, 90, 100]	[-2, 2, 0, 1, -1, 3]
8	14	&bb[0]	&a	&bb[5]	&c	&bb[0]	[50, 60, 70, 80, 90, 100]	[-2, 2, 0, 1, -1, 3]
9	15	&bb[0]	&a	&bb[5]	&c	&bb[0]	[50, 60, 70, 80, 90, 100]	[100, 2, 0, 1, -1, 3]
10	14	&bb[0]	&a	&bb[5]	&c	&bb[1]	[50, 60, 70, 80, 90, 100]	[100, 2, 0, 1, -1, 3]
11	15	&bb[0]	&a	&bb[5]	&c	&bb[1]	[50, 60, 70, 80, 90, 100]	[100, 60, 0, 1, -1, 3]
12	14	&bb[0]	&a	&bb[5]	&c	&bb[2]	[50, 60, 70, 80, 90, 100]	[100, 60, 0, 1, -1, 3]
13	15	&bb[0]	&a	&bb[5]	&c	&bb[2]	[50, 60, 70, 80, 90, 100]	[100, 60, 80, 1, -1, 3]
14	14	&bb[0]	&a	&bb[5]	&c	&bb[3]	[50, 60, 70, 80, 90, 100]	[100, 60, 80, 1, -1, 3]
15	15	&bb[0]	&a	&bb[5]	&c	&bb[3]	[50, 60, 70, 80, 90, 100]	[100, 60, 80, 70, -1, 3]
16	14	&bb[0]	&a	&bb[5]	&c	&bb[4]	[50, 60, 70, 80, 90, 100]	[100, 60, 80, 70, -1, 3]
17	15	&bb[0]	&a	&bb[5]	&c	&bb[4]	[50, 60, 70, 80, 90, 100]	[100, 60, 80, 70, 90, 3]
18	14	&bb[0]	&a	&bb[5]	&c	&bb[5]	[50, 60, 70, 80, 90, 100]	[100, 60, 80, 70, 90, 3]
19	15	&bb[0]	&a	&bb[5]	&c	&bb[5]	[50, 60, 70, 80, 90, 100]	[100, 60, 80, 70, 90, 50]
20	14	&bb[0]	&a	&bb[5]	&c	&bb[6]	[50, 60, 70, 80, 90, 100]	[100, 60, 80, 70, 90, 50]
21	18	&bb[0]	&a	&bb[5]	&c	&bb[6]	[50, 60, 70, 80, 90, 100]	[100, 60, 80, 70, 90, 50]
22	20	&bb[0]	&a	&bb[5]	&c	&bb[6]	[50, 60, 70, 80, 90, 100]	[100, 60, 80, 70, 90, 50]

Aufgabe 5.8: l oder r Auswertung

Betrachten Sie folgenden Ausschnitt eines C Programms:

```
1     void foo() {
2       int *w;
3       int **y;
4       int x[2];
5       int z;
6       z = 13;
7       w = x;
8       y = &w;
9       *x = z;
10      *(x+1) = 42;
11      z = *w;
12    }
```

1. Notieren Sie sich für alle Ausdrücke und Teilausdrücke, ob diese L-ausgewertet oder R-ausgewertet werden.
2. Notieren Sie alle Ausdrücke, die L-ausgewertet werden können. Welche Ausdrücke können R-ausgewertet werden?
3. Vervollständigen Sie folgendes Ausführungsprotokoll, wobei Sie zu jeder Zeile die L- und R-Auswertung kurz in Worte fassen.

Zeile	w	x		y	z	
2						
3	?					Initialisierung von w
4	?			?		Initialisierung von y
5	?	?	?	?		Initialisierung von x
6	?	?	?	?	?	Initialisierung von z
7	?	?	?	?	13	Die L-Auswertung der bereits definierten Variable z liefert die Adresse des Behälters von z. Die R-Auswertung der Zahlenkonstante 13 liefert den Wert 13. Damit wird die Zahl 13 in den Behälter von z geschrieben.
8	`&x[0]`	?	?	?	13	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Lösung

1.
   - z = 13: z wird L-ausgewertet und 13 wird R-ausgewertet
   - w = x: w wird L-ausgewertet und x wird R-ausgewertet
   - y = &w: y wird L-ausgewertet, &w wird R-ausgewertet und w wird L-ausgewertet
   - *x = z: *x wird L-ausgewertet, x wird R-ausgewertet und z wird R-ausgewertet
   - *(x+1) = 42: *(x+1) wird L-ausgewertet, x wird R-ausgewertet, (x+1) wird R-ausgewertet und 42 wird R-ausgewertet
   - z = *w: z wird L-ausgewertet, *w wird R-ausgewertet und w wird R-ausgewertet
2. w, y, z, *w, *x, *(x+1) sind alle L-auswertbaren Ausdrücke, die im Programm vorkommen. Alle Ausdrücke können R-ausgewertet werden. x ist ein Array, deshalb in diesem Programm nirgendwo L-auswertbar[1].

[1] Arrays in C funktionieren formell betrachtet etwas seltsam. Entgegen weitverbreitetem Irrglaube sind sie nämlich **nicht** einfach Zeiger auf das erste Element. Array-Werte sind zwar L-auswertbar, werden aber bei Verwendung im Program fast immer in einen Pointer auf das erste Element konvertiert, welcher dann nicht mehr L-auswertbar ist. Vergleiche C11 Standard, Satz 6.1.2.3 (3).

3.

Zeile	w	x		y	z	
2						
3	?					Initialisierung von w
4	?			?		Initialisierung von y
5	?	?	?	?		Initialisierung von x
6	?	?	?	?	?	Initialisierung von z
7	?	?	?	?	13	Die L-Auswertung der bereits definierten Variable z liefert die Adresse des Behälters von z. Die R-Auswertung der Zahlenkonstante 13 liefert den Wert 13. Damit wird die Zahl 13 in den Behälter von z geschrieben.
8	`&x[0]`	?	?	?	13	Die L-Auswertung der bereits definierten Variable w liefert die Adresse des Behälters von w. Die R-Auswertung der definierten Variable x liefert einen Array-Wert, welcher in die Adresse des ersten Elementes der Reihung, an die x gebunden ist, konvertiert wird. Diese Adresse wird in den Behälter von w geschrieben.
9	`&x[0]`	?	?	`&w`	13	Die L-Auswertung der bereits definierten Variable y liefert die Adresse des Behälters von y. Der Ausdruck w ist L-auswertbar: Die R-Auswertung des Ausdrucks &w liefert die L-Auswertung von w als Wert zurück, was die Adresse des Behälters von w liefert.
10	`&x[0]`	13	?	`&w`	13	Die L-Auswertung des Ausdrucks *x ist das Ergebnis der R-Auswertung von x, was durch Konvertierung die Anfangsadresse der zugehörigen Reihung ist. Die R-Auswertung der definierten Variable z liefert den Inhalt des Behälters, welcher an der Adresse liegt, die man durch die L-Auswertung von z erhält. Dies ist der Wert 13. Damit wird die Zahl 13 in den ersten Behälter der x zugehörigen Reihung geschrieben.
11	`&x[0]`	13	42	`&w`	13	Die L-Auswertung des Ausdrucks *(x+1) ist das Ergebnis der R-Auswertung von (x+1), was die Anfangsadresse + sizeof(int) und damit der zweite Behälter der zu x gehörenden Reihung ist. Die R-Auswertung der Zahlenkonstante 42 liefert den Wert 42. Damit wird die Zahl 42 in den zweiten Behälter der zu x gehörenden Reihung geschrieben.
12	`&x[0]`	13	42	`&w`	13	Die L-Auswertung der bereits definierten Variable z liefert die Adresse des Behälters von z. Der Ausdruck *w ist L-auswertbar und gibt die R-Auswertung von w, was der Inhalt des Behälters von w, also die Anfangsadresse der zu x gehörenden Reihung zurück. Die R-Auswertung eines L-auswertbaren Ausdrucks liest den Inhalt des Behälters, dessen Adresse die L-Auswertung liefert. Damit wird der Inhalt des ersten Behälters der zu x gehörenden Reihung in den Behälter von z geschrieben.

Aufgabe 5.9: C Typisierung

Entscheiden Sie für folgende Statements, ob diese valide sind.

```
1  int i, b;
2  char c = 8;
3  c = i;
4  void x = 2;
5  char* d = i;
6  d = "Typisierung macht Spass";
7  i = d[0] + 2;
8  d = (1==2);
9  int* x = 0;
10 char* y = 1;
11 d = x | y;
12 void* e;
13 x = (int*) e;
14 int a = (b = 3) + 1;
15 int f = (int)("The End?"[0]);
```

Lösung

Zeile	Gültig	Begründung
1	ja	Variablen Deklaration
2	ja	Variablen Deklaration und Definition
3	ja	Variablen Zuweisung
4	nein	void Typen kann nichts zugewiesen werden
5	ja	Variablen Deklaration und Definition
6	ja	Zuweisung von String zu char-pointer
7	ja	Zuweisung eines Ergebnisses einer Expression
8	ja	Zuweisung eines Ergebnisses eines Vergleiches
9	ja	Variablen Deklaration und Definition
10	ja	Variablen Deklaration und Definition
11	nein	Binärer OR-Operator nicht anwendbar auf int* und char*
12	ja	Variablen Deklaration
13	ja	Zuweisung nachdem gecasted wurde
14	ja	Variablen Deklaration und Definition a wird 4 zugewiesen und b 3
15	ja	Variablen Deklaration und Definition f wird das Ergebnis des Casts von „T“ zu einem integer zugewiesen

Aufgabe 5.10: Palindrom

Palindrome sind Zeichenketten, die von vorn und von hinten gelesen dasselbe ergeben. Schreiben Sie ein Programm, welches prüft, ob ein C-String, der als `char`-Pointer übergeben wird, ein Palindrom ist. Ihr Programm soll im positiven Fall 1 zurückgeben und ansonsten 0.
Hinweis: Versuchen Sie als erstes zu berechnen wie lang die Zeichenkette ist.

Lösung

Idee: String erst reversieren und dann den reversierten mit dem originalen String vergleichen.

```c
int palindrom(char* s){

  //Laenge des Strings berechnen (exklusive der 0 Terminierung)
  char* cpy_s = s;
  int len = 0;
  while(*cpy_s != 0){
    cpy_s++;
    len++;
  }

  //reversiere String
  char rev[len+1];
  for(int i=0; i < len; i++){
    rev[i] = s[len-1-i];
  }
  rev[len] = 0;

  //Strings vergleichen
  for(int i=0; i < len; i++){
    if(rev[i] != s[i]){
      return 0;
    }
  }
  return 1;
}
```

Idee: String gleichzeitig von vorne `s[i]` und von hinten `s[len-1-i]` durchlaufen.

```c
int palindrom2(char* s){

  //Laenge des Strings berechnen
  char* cpy_s = s;
  int len = 0;
  while(*cpy_s != 0){
    cpy_s++;
    len++;
  }

  //Strings vergleichen
  for(int i=0; i < len; i++){
    if(s[i] != s[len-1-i]){
      return 0;
    }
  }
  return 1;
}
```