Imperative Programmierung

Übung 7: Zeiger

Justin Kreikemeyer
Informatik, Uni Rostock

Fragen?

Leitfragen

- Rückblick: Was ist eine Variable?
- Was ist der Unterschied zwischen Wert und Adresse einer Variablen?
- Wie arbeitet man mit Adressen in C?
- In welchen Fällen arbeitet man mit Adressen in C?

Empfehlung

- Programmieren lernt man nur durch selber Programmieren!
- Löst auch nach der Übung weiter so viele Programmieraufgaben, wie irgendwie möglich ($\rightarrow$ Praktikum/Selbststudium)
- Hier: Überblick (sehr viel auf einmal!), Lösen einiger Aufgaben, Klärung von Fragen

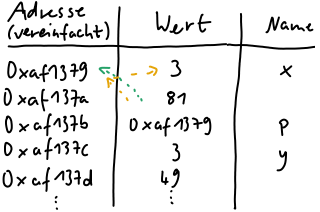
Erinnerung: Grundstruktur

```c
#include <stdio.h>
int main(int argc, char* argv[]) {  // <- char* argv = Zeiger auf Speicher!

  // Programmcode steht hier.
  int x;
  scanf("%d", &x);                  // &x ist ein Zeiger auf den Speicherbereich
                                    // von x!

  return 0;
}
```

Konzept: Zeiger/Pointer

- Variable = Name für Speicherbereich; steht stellvertretend für abgelegten Wert
- Zeiger = Name für Speicherbereich; steht stellvertretend für *Speicheradresse*
- Nützlich für
  - call-by-reference
  - Speicher wiederverwenden statt Inhalt kopieren
  - “Mehrere Rückgabewerte” bei Funktionen
  - (mehrdimensionale) Arrays

```
int  x = 3;
int* p = &x;
int  y = *p;
```



Zeiger in C

- `<typ> "*" <name> [ "=" <initialisierung> ] ";"`
- Zugriff auf Adresse einer Variablen mit `&`-Operator
- Zugriff auf Wert an einer Adresse mit `*`-Operator; `(*x)` kann genau so verwendet werden wie eine “normale” Variable (Klammerung vorbeugend möglicher Verwechselung mit Multiplikation)

```
int x = 42;        // Deklaration Variable x
int* ref_1;        // Deklaration Pointer ref_1
ref_1 = &x;        // Zugriff auf Adresse von x mittels &-Operator
int* ref_2 = &x;   // Deklaration und Zuweisung Pointer ref_2
int y = (*ref_1); // Zugriff auf Wert an Adresse mittels *-Operator
printf("%d %d %p %p", x, y, ref_1, ref_2);
*ref_1 = 97;       // Änderung des Wertes an Adresse mittels *-Op.
printf("%d %d %p %p", x, y, ref_1, ref_2);
```

Zeiger als Funktionsargumente

```c
void foo(int y) {
  y = 97;
}

int main(){
  int x = 42;
  printf("%d", x);
  foo(x);
  printf("%d", x);
  return 0;
}
```

Was gibt das obige Programm aus und warum?

Zeiger als Funktionsargumente

```c
void foo(int* y) {
  *y = 97;          // das * ist hier wichtig!!!
}

int main(){
  int x = 42;
  printf("%d", x);
  foo(&x);
  printf("%d", x);
  return 0;
}
```

Was gibt das obige Programm aus und warum?

Arrays vs. Zeiger in C

- Arrays ≈ Zeiger auf erstes Element
- Genauer: bei der Übergabe an Funktionen “zerfallen” Arrays zum Zeiger auf das erste Element
- Deshalb gibt dann auch `sizeof(arr)` nur die größe eines Zeigers zurück (und nicht des Arrays)!
- Name des Arrays kann immer wie Zeiger verwendet werden
- Pointer-Arithmetik: Inkrement mit Schrittgröße je nach Datentyp (`int`, `double`, ...)

```c
int arr[3] = {1, 2, 3};
int* arr_ref = arr;       // arr_ref zeigt auf erstes Element
int first = *arr_ref;     // Wert an arr_ref ist erstes Element
int second = *(arr + 1); // gleichbedeutend mit arr[1]; +1 bedeutet +sizeof(int)
// ...
// Beispiel: alternative Übergabe von Arrays an Funktionen
void mul(int vec1[], int size1, int vec2[], int size2) { }
void mul(int* vec1, int size1, int* vec2, int size2) { }
```

Mehrdimensionale Arrays

- Array1 = Zeiger auf erstes Element Array1
- Erstes Element Array1 = Zeiger auf erstes Element Array2
- Erstes Element Array2 = . . .
- Erfordert idR Arbeit mit dynamischem Speicher, wenn variable Länge gewünscht; für heute Ausnutzung des C99 Standards für variable Array-Längen:

```
void printArr2d(int dim1, int dim2, int arr[dim1][dim2]) {
  for (int i = 0; i < dim1; ++i) {
    for (int j = 0; j < dim2; ++j) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
}
// ...
int multi[3][3] = { {0, 1, 2}, {3, 4, 5}, {6, 7, 8} };
printArray2d(3, 3, arr);
```

Gemeinsames Beispiel

```
1  int main(){
2    int a = 1;
3    int x = 7;
4    int y[2] = {5, 6};
5    int* p = &x;
6    int z = (*p) * (*(y + a));
7    return 0;
8  }
```

Skizzieren Sie den Speicherzustand nach jeder Anweisung!

Fragen?

Aufgaben: Zeiger in C

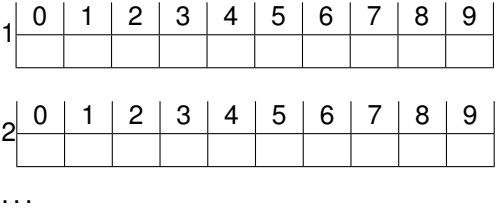
Bearbeiten Sie die folgenden Aufgaben[1]! Nutzen Sie dazu die Konzepte aus dieser Übung.
Bearbeitungszeit: bis 10 Minuten vor Schluss. Dann Besprechung von häufigen Problemen.

[1] Weitere Aufgaben können jederzeit beim Übungsleiter erfragt werden.

Speicherzustand Verstehen I

Gegeben sei das folgende Programm, welches auf einem Computer mit 10 Speicherzellen ausgeführt wird. Jede Zelle kann entweder eine ganze Zahl oder die Adresse (0-9) einer anderen Speicherzelle (=Zeiger) enthalten. Neu deklarierte Variablen reservieren immer die erste freie Speicheradresse. Skizzieren Sie den Zustand des Programms nach jeder Anweisung!

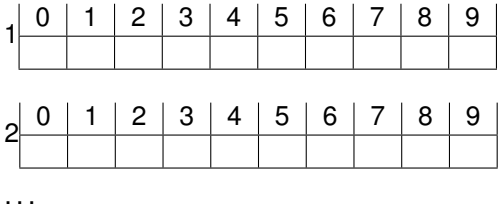
```
1  int i = 3;
2  int j[3] = {0};
3  int* x = &i;
4  int* y = j + 1;
5  *(y + 1) = 7;
6  int** z = &y;
```



Speicherzustand Verstehen II

Wie auf vorheriger Folie; neues Programm:

```
1  int a = 1;
2  int* b = &a;
3  int c[3] = {1, 2, 3};
4  int* d = c + 1;
5  *(d + a) = 6;
```



Programmieren mit Zeigern

MaxPointer Schreiben Sie eine Funktion, welche zwei Zeiger auf ganze Zahlen übergeben bekommt und den Zeiger zurückgibt, hinter dem sich der größere der beiden Werte verbirgt.

MinMax Schreiben Sie eine Prozedur, welche das Maximum und das Minimum einer Liste an ganzen Zahlen bestimmt und geben Sie mit Hilfe von Zeigern das Maximum und das Minimum zurück.

Programmieren mit Mehrdimensionalen Arrays: TicTacToe

1. Deklarieren Sie ein zweidimensionales Array `board` der größe 3×3
2. Schreiben Sie eine Funktion `void reset(int dim1, int dim2, int board[dim1][dim2])`, welche alle Elemente des Arrays auf -1 setzt.
3. Schreiben Sie eine Funktion `void print(int dim1, int dim2, int board[dim1][dim2])`, welche das Spielfeld formatiert ausgibt. Ist ein Eintrag -1, soll - ausgegeben werden; Ist er 0, soll `x` ausgegeben werden; Ist er 1, soll `o` ausgegeben werden.
4. Schreiben Sie eine Funktion `void place(int player, int x, int y, int dim1, int dim2, int board[dim1][dim2])`, welche an der Koordinate (x, y) im Spielfeld das Zeichen des jeweiligen Spielers markiert. Achten Sie auf die Einhaltung der Grenzen.

Testen Sie ihre Funktionen jeweils in der Main-Funktion!