LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Grundlagen C und C++

Einheit 02: Grundlagen in C

Lorenz Schauer
Lehrstuhl für Mobile und Verteilte Systeme

Heutige Agenda

Grundlagen in C

- Arrays und Zeiger
- #define Direktive
- Typdefs und Enums

Auf zur Objektorientierung:

- Strukturen (struct)
  - Deklarieren, Erzeugen und initialisieren
  - Referenzieren mit Pointer

Übungen

- Zahlenraten
- Pointer
- Nutzereingaben

Lernziele

- Weitere Grundlagen zu C kennenlernen
- Mit den Grundlagen umgehen können
- Übungen mit C Grundlagen

Aufgabe

Schreiben Sie nun ein weiteres C-Programm mit dem Namen *ZahlenRaten.c*, das folgende Aufgaben übernimmt:

- Ihr Programm denkt sich zunächst eine Zahl zwischen 0 und 10 aus
  - Hinweise:
    - rand() liefert eine ganzzahlige Zufallszahl im Bereich von 0 und 2^15 - 1.
    - srand() wird zur Initialisierung des Zufallszahlengenerators benutzt
    - Mit dem Modulo Parameter % können wir den Zufallszahlbereich eingrenzen
- Ihr Programm fordert nun den Nutzer immer wieder auf eine Zahl auf der Konsole einzugeben, bis der Nutzer die vom Programm ausgedachte Zahl korrekt erraten hat!
  - Gibt der Nutzer also eine falsche Zahl ein, meldet das Programm zurück, dass die eingegebene Zahl falsch ist und der Nutzer wird erneut aufgefordert eine Zahl einzugeben
  - Gibt der Nutzer hingegen die vom Programm ausgedachte Zahl ein, meldet das Programm „Herzlichen Glückwunsch“ und beendet die Ausführung

Arrays und Zeiger

Arrays speichern mehrere Daten des gleichen Typs

- Sind ein Verweis auf eine Folge von Variablen (Fächer)
- Syntax sehr ähnlich zu Java

```
// Deklaration:
datentyp name[sizeOfArray];

//Bsp.:
float messwerte[10];

// Initialisierung:
name[index] = Wert;

//Bsp.:
messwerte[0] = 23.0;
messwerte[1] = 22.2;
messwerte[2] = 21.7;
messwerte[3] = 20.9;
messwerte[4] = 20.5;
```

Arrays deklarieren und initialisieren in einem Schritt:

```
int punkte[5] = { 1, 3, 5 }; // Restlichen Werte werden auf 0 gesetzt.

// Oder implizit:

int punkte[] = {6,2,5}; // Größe entspricht 3
```

Arrays durchlaufen:

```
int punkte[5], i;

// Werte setzen
for(i=0; i<5; i++) {
   punkte[i] = i+1;
}

// Werte auslesen
for(i=0; i<5; i++) {
   printf("(Index %d) Punktzahl Aufgabe %d: %d\n", i, i+1, punkte[i]);
}
```

Mehrdimensionale Arrays

Auch mehrdimensionale Arrays möglich:

- Wieder analog zu Java
- Bsp.:
  - `int brett[8][8];`
  - `double nochMehrDimentsionen[10][5][2];`

```
int brett[8][8] = { 0 };

brett[2][1] = 1;
brett[4][2] = 2;
brett[3][5] = 3;
brett[6][7] = 4;

int i, j;

// Schleife fuer Zeilen, Y-Achse
for(i=0; i<8; i++) {

   // Schleife fuer Spalten, X-Achse
   for(j=0; j<8; j++) {
      printf("%d ", brett[i][j]);
   }

   printf("\n");
}
```

```
// AUSGABE:

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 3 0 0
0 0 2 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 4
0 0 0 0 0 0 0 0
```

Zeiger (Pointer)

Ein Array besteht im Prinzip nur aus Zeigern, welche auf die zum Array gehörenden Variablen zeigen.

- Der Zugriff auf die Variablen erfolgt also mittels Zeiger (Pointer)

Was sind Pointer bzw. Zeiger?

- Ein Zeiger repräsentiert eine Adresse und nicht wie eine Variable einen Wert.
- Will man auf den Wert der Adresse zugreifen, auf die ein Zeiger zeigt, muss der Stern * vor den Namen gesetzt werden.

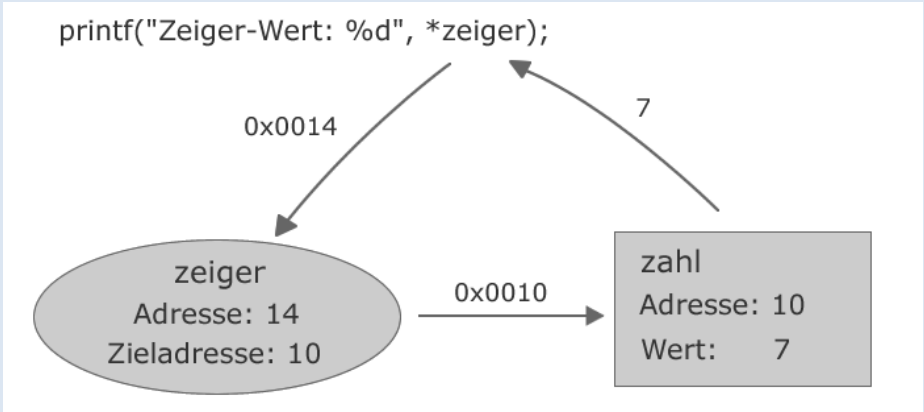
```
// Beispiel:

int zahl = 7;

int *zeiger;

zeiger = &zahl;

printf("Zeiger-Wert: %d\n", *zeiger);
```

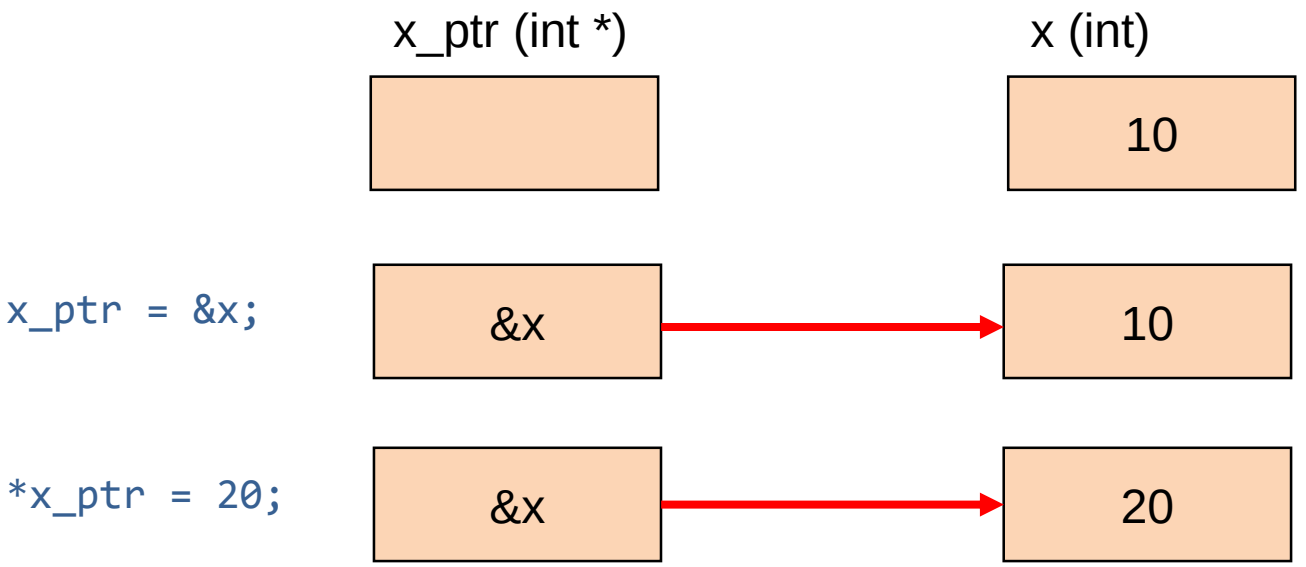


Quelle: C-HowTo

C: Adressen und Zeiger

```
int x = 10; // Variable x mit Wert 10
&x; // Adresse von x Bsp.: 0x000001

int *x_ptr; // Pointer vom Typ int *
&x_ptr; // Adresse von x_ptr Bsp.: 0x000034221
```



C: Adressen und Zeiger Beispiel


```c
#include <stdio.h>
int main(int argc, const char *argv[])
{
   int i = 1337;
   int *addressOfI; //Pointer vom Typ 'int *'
   addressOfI = &i; //Referenzübergabe: Übergeben der Adresse von i

   printf("address of i: %p\n", addressOfI); //address of i: 0xbffff738
   printf("value of i: %d\n", i); //value of i: 1337

   addressOfI = 10;
   //Error: Incompatible Integer to Pointer Conversion Assigning to 'int *'
   from 'int'

   *addressOfI = 10;
   printf("value of i: %d\n", *addressOfI); //value of addressOfI: ?
   printf("value of i: %d\n", i); //value of i: ?
   return 0;
}
```

Arrays und Zeiger

Da Arrays im Prinzip eben aus Zeigern bestehen, bedienen wir uns nun der Zeigerarithmetik, um Arrays zu durchlaufen und zu lesen bzw. zu schreiben:

```
// Beispiel:

double d_array[] = {23.3, 22.1, 33.5, 45.2};

// Zeiger: Zeigt auf den Anfang eines Arrays:
double* pos = d_array;

// Alternativ auch:
double* pos = &d_array[0];

printf("Wert 1: %.2f",*pos); // Ausgabe: Wert 1: 23.30

// Zeiger weiterwandern lassen:
pos++;

printf("Wert 2: %.2f",*pos); // Ausgabe: Wert 2: 22.10
```

Arrays durchlaufen mittels Zeiger:

```
// Beispiel:

 double d_array[] = {23.3, 22.1, 33.5, 45.2};

 double* pos = d_array;

 int i;
 for(i=0;i<4;i++,pos++){
    printf("Wert %d: %.1f\n",i,*pos);
 }

//Ausgabe:

Wert 0: 23.3
Wert 1: 22.1
Wert 2: 33.5
Wert 3: 45.2
```

Arrays: Größe bestimmen

Für Schleifen brauchen wir die Größe des Arrays, damit wir nicht über den Allokierten Speicherbereich rauslaufen!

- Ähnlich zu Java: `array.length`
- In C: Funktion `sizeof(data)`
  - Liefert die Größe von `data` in Bytes zurück!
  - Für Anzahl der Elemente muss durch die Bytezahl des Datentyps geteilt werden

```
//Beispiele:

double d_array[] = {23.3, 22.1, 33.5, 45.2};

int groesse_in_bytes = sizeof(d_array); // Hier 32, da 4*8 Byte für double!

int anzahl_der_elemente = sizeof(d_array)/sizeof(double); // Hier: 4

//Damit könnte for-Schleife durchlaufen werden
int i;
for(i=0;i< anzahl_der_elemente; i++){…}
```

Die `#define` Direktive

Mittels der `#define` Direktive können für den Präprozessor:

- Symbolische Konstanten
  - Bsp.: `#define NMAX 100;`
- Oder Makros definiert werden
  - Bsp.: `#define SQR(a) (a*a);`

- Erleichtern das Programmieren durch Wiederverwendung
- Bsp.: Unsere length-Funktion für Arrays als Makro

```
#define length(x) (sizeof(x)/sizeof(x[0]))

int main(){

//…
 for(i=0;i<length(d_array); i++){…}

}
```

Typdefinitionen: typedef

Das Schlüsselwort `typedef` erlaubt es uns eigene Namen für Typen zu vergeben.

- Ähnlich zur `#define` Direktive, aber:
  - Wird vom Compiler verwertet und nicht vom Präporzessor!
  - Beschränkt sich auf die Vergabe von symbolischen Namen auf Typen, nicht auf Werte, wie bei `#define`

```
// Beispiel

typedef unsigned char BYTE; // Laut Konvention Großbuchstaben

// Aber auch möglich in Kleinbuchstaben:
typedef unsigend char byte;

int main(){


   // Verwendung:
   BYTE b1 = 65;

   //…
}
```

Aufzählungstypen: enum

Enumerations (enum) erlauben uns Aufzählungstypen zu definieren:

- Integer-Variablen, die einen bestimmten Wert enthalten:
  - Bsp.: `false=0, true=1`
- Diese Werte können dann über Namen angesprochen werden
  - Bsp.: Wochentage von 0 bis 6 über „So“ bis „Sa“

```
// Definition eines enum-Types:

enum Wochentag{So,Mo,Di,Mi,Do,Fr,Sa};
enum Richtung{NORD, OST=90, SUED=180, WEST=270};
enum bool{false,true};

// Ansprechen über Namen. Bsp.:

printf(„Die aktuelle Richtung ist: %d“, SUED); // Gibt 180 aus

Wochentag heute = Mo;
printf(„Der aktuelle Wochentag ist: %d“, heute); // Gibt 1 aus.

while(true){…} // Endlosschleife
```

Enums über typedef definieren

Man kann sich also mittels `typedef` und `enum` seinen eigenen globalen Typen definieren:

- Bsp.: `Boolean`

```
enum bool {
   false, true
};

typedef enum bool boolean;

int main(){

   boolean zahl_erraten = false;

   if(zahl_erraten){
      puts(„Zahl wurde erraten…“);
   }
   else{
      puts(„Bitte weiter raten!“);
   }
}
```

C-Strukturen: `struct`

Daten befinden sich zusammenhangslos im Speicher.

Will man Daten (Variablen) sinnvoll bündeln, können Strukturen (`structs`) verwendet werden:

- Ähnlich zu gekapselten Daten in Objekten
- Verwendung des Schlüsselworts: `struct`

```
// Beispiel Adresse:

struct adresse {
   char name[50];
   char strasse[100];
   short hausnummer;
   long plz;
   char stadt[50];
};
```

Mit Strukturen arbeiten

Erzeugen eines Objekts bzw. einer Variablen der Struktur:

- Bsp.: `struct adresse meine_adresse;`

Der Zugriff auf die Elemente der Struktur:

- Punktnotation (ähnlich wie in Java)
  - Bsp.: `meine_adresse.plz = 34;`
- Ganze Zeichenketten können mittels `strcpy` ans Array übergeben werden
  - Bsp.: `strcpy(meine_adresse.name, „Hansi Maier“);`

```
// Variable der Struktur erstellen
struct adresse adresseKurt;

// Zugriff auf die Elemente
strcpy(adresseKurt.name, "Kurt Kanns");
strcpy(adresseKurt.strasse, "Kannichweg");
adresseKurt.hausnummer = 23;
adresseKurt.plz = 45678;
strcpy(adresseKurt.stadt, "Kannstadt");

printf("Name: %s\n", adresseKurt.name);
```

Strukturen initialisieren

Erzeugung und Initialisierung zusammen mit der Deklaration möglich:

```
// Beispiel 1: Erzeugung bei Deklaration und dann initialisieren

struct person {
   char name[50];
   int alter;
} kurt, dieter, antonia;

kurt.alter = 33;
dieter.alter = 27;
antonia.alter = 23;

// Oder gleich initialisieren:

struct person {
   char name[50];
   int alter;
}
kurt = { "Kurt Kanns", 33 },
antonia = { "Antonia", 23 };
```

Structs über typedef definieren

Wir können auch wie vorher Strukturen mit tyedef definieren und diese dann einfach als eigenen Namen ansprechen

```
typedef struct {
   char name[50];
   int alter;
}Person;

int main(){

   //Erzeugen und Initialisieren:
   Person hansi = {„Hansi“,23};

   //Nur Erzeugung:
   Person peter;

   // Dann initialisieren:
   strcpy(peter.name,"Peter");

   printf("Die Person heisst: %s\n",hansi.name);
   printf("Die Person heisst: %s\n",peter.name);

   return 0;
}
```

Mittels Pointer auf Strukturen arbeiten

Wir können Strukturen auch mittels Zeiger erzeugen (referenzieren)

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

typedef struct {
   char name[50];
   int alter;
}Person;

void ausgabePerson(Person *person) {
   printf("Name: %s\n", (*person).name); //alternativ: person->name
   printf("Alter: %d\n", person->alter);
}

int main() {
   Person kurt;
   strcpy(kurt.name, "Kurt Kanns");
   kurt.alter = 33;
   ausgabePerson(&kurt); //Adresse Übergeben
   return 0;
}
```

Programmieraufgabe

Aufgabe: Nutzereingaben verarbeiten

Schreiben Sie ein C-Programm, das folgenden Aufgaben erledigt:

- Der Benutzer wird immer wieder aufgefordert eine Zahl einzugeben, und zwar solange, bis er eine 0 eingibt.
  - Achten Sie darauf, dass der Benutzer mit seinen Eingaben nicht in einen fremden Speicherbereich schreiben kann
- Die eingegebenen Zahlen sollen dann zur Kontrolle auf der Konsole ausgegeben werden
- Daraufhin berechnet das Programm den Durchschnitt, die kleinste und die größte eingegebene Zahl.
- Die Ergebnisse werden dem Benutzer dann auf der Konsole ausgegeben.

Nutzen Sie für die Bearbeitung der Aufgabe die eben vorgestellten C-Grundlagen