Datentypen: Enum, Array, Struct, Union

C-Kurs 2014, 2. Tutorium

Freitagsrunde

http://wiki.freitagsrunde.org

17. September 2014

Inhaltsverzeichnis

Aufzählungen (Enums)

```
1 int
```

```
1 double
```

```
1 enum day_t {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

- Neu definierter Datentyp: **enum day_t** mit 7 möglichen Werten
- Der Compiler wandert die Elemente des *enum* in Integer um

Enum

```
enum day_t {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
enum day_t current_day = Tue;

switch (current_day) {
  case Sat:
  case Sun:
    printf("Es ist Wochenende! \n");
    break;
  default:
    printf("Es ist unter der Woche. \n");
    break;
}
```

Enum

- werden auch als Aufzählungstypen bezeichnet
- Intern als Integer gehandelt
- Elemente sind nacheinander durchnummeriert
- Beispielsweise lassen sich die Wochentage gut repräsentieren

Boolean repräsentiert durch Enums

- In C entspricht **false** dem Wert 0
- alles ungleich 0 ist **true**

```
enum bool {FALSE,TRUE};
```

Wenn man nichts angibt, wird das erste Element 0, das zweite 1, usw
Man kann auch diese Konstanten direkt einen Wert explizit zuweisen:

```
enum bool {
  TRUE  = 42,
  FALSE = 0
};
```

Enum-Boolean mit typedef

- Kann wie folgt verwendet werden

```
enum bool {FALSE, TRUE};
enum bool flag = TRUE;
```

- Sieht jedoch ungewöhnlich aus
- Besser ist die Verwendung mit **typedef**

```
typedef enum bool {FALSE, TRUE} boolean;
boolean flag = TRUE;
```

Colors mit enum

Schreibe eine Datenstruktur Color, die die Grundfarben **black**, **white**, **red**, **yellow**, **green**, **cyan**, **blue**, und **magenta** definiert, indem den einzelnen Varianten explizit HEX-Werte zugewiesen werden!

```
1 typedef enum color {
2     ...
3     ...
4 } Color;
```

Lösung der Übungsaufgabe

```
1  typedef enum color{
2    WHITE   = 0xFFFFFF,
3    BLACK   = 0x000000,
4    RED     = 0xFF0000,
5    YELLOW  = 0xFFFF00,
6    GREEN   = 0x00FF00,
7    CYAN    = 0x00FFFF,
8    BLUE    = 0x0000FF,
9    MAGENTA = 0xFF00FF
10 } Color;
11
12 //Verwendung:
13 printf("White: %d(decimal) %x(hex)\n", WHITE, WHITE);
```

Inhaltsverzeichnis

Java-Arrays vs. C-Arrays

- Arrays in C sind ähnlich zu Arrays in Java
- Länge in **Bytes** kann mit **sizeof** abgefragt werden
- Für echte Länge: Division mit der Größe eines Elements des Arrays
  → ebenfalls mit **sizeof**

```
1 Color rainbow[] = {RED, YELLOW, GREEN, CYAN, BLUE,
    MAGENTA};
2 int length = sizeof(rainbow)/sizeof(Color);
3
4 //Verwendung:
5 printf("Der Regenbogen hat %d Farben\n", length);
```

C-Arrays im Detail

- Werte eines Arrays werden hintereinander im Speicher abgelegt
- Natürlich ist es in C genauso wie in Java möglich mehrdimensionale Arrays anzulegen
- Array selbst ist nur die Adresse des ersten Elements
- Zur Laufzeit weiß das C-Programm nicht mehr, dass es ein Array ist
  - Bei **sizeof** legt der Compiler eine Tabelle an
  - Diese wird nach dem Übersetzen gelöscht
  - Dort ist unter dem **Namen** die Länge des Arrays vermerkt

```
1 Color rainbow[] = {RED, YELLOW, GREEN, CYAN, BLUE,
    MAGENTA};
2 int length = sizeof(rainbow)/sizeof(Color);
```

Inhaltsverzeichnis

struct

```
1 struct name {
2     datentyp1 var1;
3     datentyp2 var2;
4     ...
5 };
```

- Benutzerdefinierter komplexer Datentyp
- Dabei verwendet man das Schlüsselwort **struct**
- Auch **typedef** möglich, ähnlich wie enum
- **struct** Konstrukt so groß, wie die Summe der Größen seinen Membervariablen
- Semikolon am Ende!

Übungsaufgabe: Verwendung von struct

Eine Handvoll Farben mit *enum* ist zu begrenzt. Wir wollen die einzelnen RGB Farbkanäle red, green und blue gezielt verändern und damit alle Farben definieren, die es gibt.

- Deklariert ein *struct* **colorDefinition** mit den „Membervariablen“ **red**, **green** und **blue** (unsigned char: 8 Bit, 0..255)
- definiert die Farben **black**, **blue** und **white** als „Objekte“.
- Gibt den R-Wert von **black** am Bildschirm aus

Lösung: struct

```
1  typedef struct colordef {
2    unsigned char red;
3    unsigned char green;
4    unsigned char blue;
5  } colorDefinition;
6
7  colorDefinition black = {0,0,0};
8  colorDefinition white = {255, 255, 255} ;
9  colordefinition blue = {.blue=255};
10
11 //Verwendung:
12 printf("%d", black.red);
```

Inhaltsverzeichnis

Union

```
union name{
  datentyp1 var1;
  datentyp2 var2;
  ...
};
```

- Wird genauso wie ein struct deklariert
- Dabei verwendet man jedoch das Schlüsselwort **union**
- Alle Variablen sind Bezeichner für dieselbe Speicherstelle
- somit ist das **union** Konstrukt nur so groß, wie seine größte Variable (platzsparend!)
- Damit können wir **colorDefinition** erweitern, sodass wir wahlweise RGB-Angaben oder HEX-Werte speichern können.

Union: Erweiterung des Color-Types

```
1  typedef union color{
2    struct rgb {
3      unsigned char red;
4      unsigned char green;
5      unsigned char blue;
6    } rgb;
7    int hex;
8  } Color;
9
10 Color schweinchenRosa = {0xffb6c1};
11 Color navy = {0, 0, 128}
12
13 //Verwendung
14 printf("B-Wert der Farbe Navy: %d\n", navy.rgb.blue);
```

Inhaltsverzeichnis

Zusammenfassung: enum, Arrays

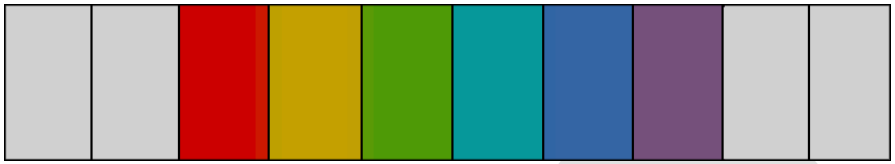
- **enum color**: Neu definierter Datentyp mit einer Auflistung von möglichen Werten.
- **typedef enum color {...} Color**: das Wort *Color* kann als Datentyp anstelle von *enum color* verwendet werden
- **Color rainbow[6]**: ein Array, das 6 Elemente vom Typ Color enthalten kann
- **Color flag[] = {BLACK, RED, YELLOW}**: ein Array, das 3 Elemente vom Typ Color enthält
- **sizeof(flag)/sizeof(Color)**: die Länge unseres Arrays (hier:3)

Zusammenfassung: struct, union

- **struct stru {int a; char b;};** Neu definierter komplexer Datentyp *struct stru* mit int und char nacheinander
- **struct stru var1 = {1, 'x'};** : Verwendung von struct, beide Variablen werden gesetzt
- **union uni {int a; char b;};** Neu definierter Datentyp *union uni*, bei dem die Elemente wahlweise int oder char enthalten können
- **union uni var2 = {.a = 1};** : Verwendung von union, nur eine Variable darf gesetzt werden

Inhaltsverzeichnis

Übungsaufgabe: Arrays wie in Java



```
Color rainbow[] = {RED, YELLOW, GREEN, CYAN, BLUE,
    MAGENTA};
int length = sizeof(rainbow)/sizeof(Color);
```

Angenommen, wir wollen eine Funktion **printColor** benutzen, welche die einzelnen Farben im Array auf dem Terminal ausgibt. Die Funktion **printColor** hat dabei folgende Signatur

```
void printColor(Color farben[], int length);
```

Übungsaufgabe: Arrays wie in Java

```
1 //Regenbogen
2 Color rainbow[] = {RED, YELLOW, GREEN, CYAN, BLUE,
    MAGENTA};
3 int length1 = sizeof(rainbow)/sizeof(Color);
4
5 //Deutsche Flagge
6 Color flag[] = {BLACK, RED, YELLOW};
7 int length2 = sizeof(flag)/sizeof(Color);
```

- Bei mehreren Arrays, die an **printColor** übergeben werden, muss man die Arrays jedes mal anlegen, die Länge ermitteln, in einer separaten Variable speichern und merken, etc... unübersichtlich!
- Wir vermissen Java-Arrays und .length!

Implementierung: Arrays wie in Java

```
1  typedef struct colorarray {
2    const int length;
3    color content[9];
4  } colorArray;
5
6  //Ein colorArray struct anlegen,
7  colorArray rainbow = {
8    .length= 9,
9    .content= {RED, YELLOW, GREEN, CYAN, BLUE, MAGENTA,
       RED, BLUE, YELLOW}
10 };
11
12 //Laenge des Arrays
13 int array_length = rainbow.length
```