Datenstrukturen in C

- Aus den einfachen Datentypen lassen sich komplexere konstruieren
- Array und `struct`: zusammengesetzte Datentypen
- `union`: Zugriff auf gleichen Speicherbereich über unterschiedliche Identifier

- (Speicher-)Objekt: Speicherbereiche eines bestimmten Typs
  - Variablen, Parameter, etc.

sizeof

Der `sizeof`-Operator

- wird wie eine Funktion genutzt
- ermittelt die Größe des als Argument übergebenen Datentyps in Byte
- kann auch auf Variablen aufgerufen werden
- ermittelte Werte plattformabhängig (Ausnahme: `sizeof(char) == 1`)

```
1 size_t size;
2 size = sizeof(char);      // --> 1
3 size = sizeof(size_t);    // --> 8
4 size = sizeof(void *);    // --> 8
5
6 size_t a = sizeof(size_t);
7 size_t b = sizeof(a);
8 // Nun gilt: a == b, da ‘a‘ vom Typ ‘size_t‘ ist.
```

(die Größenangaben sind Beispielwerte auf einem üblichen LP64-System)

Alignment

- Anforderung an Ausrichtung der Speicheradresse eines Objektes
  Speicheradresse muss ein Vielfaches vom Alignment sein
- Abhängig vom Datentyp
- Größe eines Datentyps ist Vielfaches des Alignments

- Beispiel: `int`
  - Größe: 4 Byte
  - Alignment entweder 1, 2, oder 4 Byte
  - Üblicherweise 4 Byte Alignment

Pointer

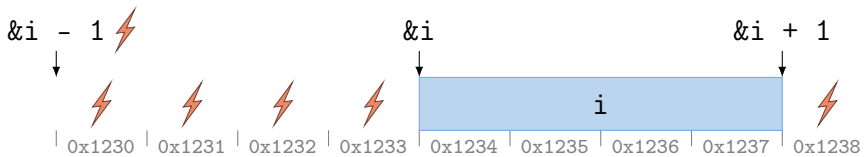
- Pointer: Adresse eines Speicherobjektes, "zeigt" auf das Objekt
- Arrays: zusammenhängender Speicherbereich, enthält Datenobjekte gleichen Typs direkt aufeinanderfolgend
- Arrays können ebenfalls über Pointer genutzt werden!

- Operator `&`: nimmt Adresse eines Objekts
- Operator `*`: Dereferenzieren eines Pointers (Speicherzugriff)

```
1 int i = 0;
2 int* i_ptr = &i;
3 int i_new = *i_ptr;    // i_new == i
```

Pointerarithmetik

```
1 // Annahme: sizeof(int) == 4
2 int i = 0;
3 int* i_ptr = &i;  // z.B. 0x1234
4 i_ptr++;          //    -> 0x1238 (= 0x1234 + 4)
5 i_ptr -= 2;       //    -> 0x1230 (= 0x1238 - 8)
6
7 // Achtung: die letzte Operation ist in diesem Fall UB, da der
8 //           resultierende Pointer nicht mehr auf ein Element
9 //           im "Array" zeigt.
```



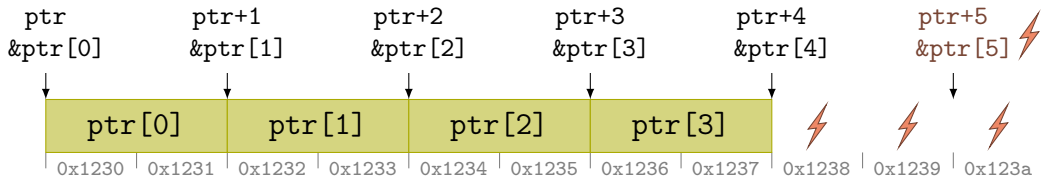
Pointerarithmetik: Arraysubscript

- `ptr[0]` $\widehat{=}$ `*ptr`
- `ptr[3]` $\widehat{=}$ `*(ptr + 3)`
- `ptr[`*index*`]` $\widehat{=}$ *index*`[ptr]`
  (Alte Syntax...)

Valide Pointer zeigen immer auf...

- ...ein Objekt (z.B. eine Variable),
- ...eine Stelle in einem Array, oder
- ...an das Ende eines Arrays
  (dann nicht dereferenzierbar)

Andernfalls: Verhalten undefiniert!



Quiz: Pointer und Pointerarithmetik (1)

Seien `ptr` und `ptr2` vom Typ `int*` und `sizeof(int)== 4`.
Welche Aussagen treffen zu?

- ☐ `ptr++` inkrementiert den Wert der Speicheradresse um 1.
- ☐ `ptr++` inkrementiert den Wert der Speicheradresse um 4.
- ☐ `ptr + ptr2` addiert die beiden Speicheradressen.
- ☐ `ptr2 - ptr` subtrahiert den Wert von `ptr` vom Wert von `ptr2`.

Quiz: Pointer und Pointerarithmetik (2)

Seien `ptr` und `ptr2` vom Typ `int*` und `sizeof(int)== 4`.
Welche Aussagen treffen zu?

- ☐ `*(ptr + 5)` ist äquivalent zu `ptr[5]`.
- ☐ `ptr & 0x2` verundet den Wert der Speicheradresse mit `0x2`.
- ☐ `ptr2 -= *ptr` zieht `4 * (*ptr)` vom Wert der Speicheradresse von `ptr2` ab.

void-Pointer und implizite Pointer-Casts

- void-Pointer nicht dereferenzierbar
- Dekrementieren oder Inkrementieren eines void-Pointers ist undefiniert
- Casting zw. void-Pointers und anderem Pointer-Typ ist implizit

```
1 int* i_ptr = /* ... */;
2 void* v_ptr = i_ptr;
3 int* i_ptr2 = v_ptr;
```

Explizite Pointer-Casts

- Explizite Typumwandlung
- Neuer Datentyp darf keine strengeres Alignment fordern
- Derenferenzierung von umgewandelten Pointern: *undefined behavior*!
  (Einzige Ausnahme: `char`-Pointer)
- Explizite Casts daher vermeiden

```
1 int* i_ptr = /* ... */;
2 char* c_ptr = (char*) i_ptr;      // Zugriff möglich
3 short* s_ptr = (short*) i_ptr;    // Zugriff undefined behavior
4 long* l_ptr = (long*) i_ptr;      // Cast undefined behavior, da
5                                   // long strengere Alignment-
6                                   // anforderungen hat als int!
```

Quiz: `void`-Pointer und Pointer-Casts (1)

Welchen Wert (dezimal) hat die Variable c
am Ende des folgenden Codeauschnittes?

```
1 int i = 42;
2 char c = *(&i);
```

- ☐ Pointer-Casts sind *undefined behavior*.
- ☐ 8
- ☐ 42
- ☐ Es kommt zu einem Compilerfehler

Quiz: `void`-Pointer und Pointer-Casts (2)

Welchen Wert (dezimal) hat die Variable `c` am Ende des folgenden Codeauschnittes?
(Annahme: `sizeof(int) == 4`, `CHAR_BIT == 8` und Little Endian)

```
1 int i = 123456789;
2 char* c_ptr = (char*) &i;
3 char c = c_ptr[2];
```

- ☐ 91
- ☐ Der Pointer-Cast ist *undefined behavior*.
- ☐ Es kommt zu einem Compilerfehler.
- ☐ Der Zugriff auf das dritte Element des Arrays, auf das `c_ptr` zeigt, ist *undefined behavior*, da das Array nur ein Element enthält.
- ☐ Die Dereferenzierung von `c_ptr` ist *undefined behavior*, weil `c_ptr` aus dem Cast eines `int`-Pointers hervorging.

Quiz: `void`-Pointer und Pointer-Casts (3)

Welchen Wert (dezimal) hat die Variable `i` am Ende des folgenden Codeausschnittes? (Annahme: `sizeof(int) == 4`, `CHAR_BIT == 8` und Little Endian)

```
1 char c = 42;
2 int* i_ptr = (int*) &c;
3 int i = *i_ptr;
```

- ☐ 8387882
- ☐ Es kommt zu einem Compilerfehler.
- ☐ 42
- ☐ Der Pointer-Cast ist *undefined behavior*.
- ☐ Die Pointer-Dereferenzierung ist *undefined behavior*.

Quiz: `void`-Pointer und Pointer-Casts (4)

Welchen Wert (dezimal) hat die Variable `i` am Ende des folgenden Codeausschnittes?

```
1 char c = 42;
2 void* v_ptr = &c;
3 int i = *v_ptr;
```

- [ ] 8387882
- [ ] Es kommt zu einem Compilerfehler.
- [ ] Es fehlt ein expliziter Cast des `char`-Pointers zum `void`-Pointer.
- [ ] 42

Quiz: `void`-Pointer und Pointer-Casts (5)

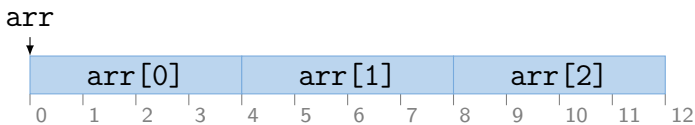
Welchen Wert (dezimal) hat die Variable `j` am Ende des folgenden Codeausschnittes? (Annahme: `sizeof(int) == 4`, `CHAR_BIT == 8` und Little Endian)

```
1  int i = 42;
2  void* v_ptr = &i;
3  int* i_ptr = v_ptr;
4  int j = *i_ptr;
```

- ☐ 8387882
- ☐ Es kommt zu einem Compilerfehler.
- ☐ Es fehlt ein expliziter Cast des `int`-Pointers zum `void`-Pointer.
- ☐ 42
- ☐ Die Pointer-Dereferenzierung ist *undefined behavior*.

Arrays: Deklaration und Definition

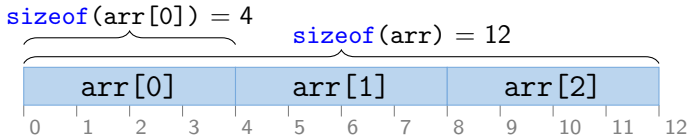
- Speicherbereich mit Datenobjekten gleichen Typs direkt aufeinanderfolgend
- Zugriff: syntaktisch analog zu Pointern (Bsp: `arr[0]` oder `*arr`)
- Deklaration mit impliziter Größenangabe:
  ```
  int arr[3];
  ```
- Deklaration und Definition in einem Statement mittels "Compound Literal":
  ```
  int arr[3] = {1, 2, 3};
  ```
- Verzicht auf explizite Größenangabe:
  ```
  int arr[] = {1, 2, 3};
  ```



Arrays: Größe

- Bestimmung der Größe durch den `sizeof`-Operator
  Achtung: gesamte Größe muss durch Elementgröße dividiert

```
1 int arr[3] = {1, 2, 3};
2 size_t arr_len = sizeof(arr) / sizeof(arr[0]);    // --> 3
```



- variable Größe:

```
1 void func(size_t size) {
2     char buf[size];
3     // ...
4 }
```

Arrays als Parameter (1)

```c
void func(int* arr, size_t arr_length) {
    for (size_t i = 0; i < arr_length; i++) {
        printf("%d\n", arr[i]);
    }
}

int main(void) {
    int arr[3] = {1,2,3};
    func(arr, sizeof(arr) / sizeof(arr[0]));
    // ...
}
```

- Achtung! `sizeof` von Array vs. `sizeof` von Pointer

Arrays als Parameter (2)

```c
void func1(int arr[arr_length], size_t arr_length) {
    for (size_t i = 0; i < arr_length; i++) {
        printf("%d\n", arr[i]);

    }
}
void func2(int arr[3]) {
    for (size_t i = 0; i < 3; i++) {
        printf("%d\n", arr[i]);
    }
}
int main(void) {
    int arr[3] = {1,2,3};
    func1(arr, sizeof(arr) / sizeof(arr[0]));
    func2(arr); // ...
}
```

Quiz: Arrays (1)

Was ist der Wert von a nach folgendem Codeauschnitt?

```
1 int arr[4] = {1,2,3};
2 int a = arr[3];
```

- ☐ 3
- ☐ 0
- ☐ Es kommt zu einem Compilerfehler

Quiz: Arrays (2)

Was ist der Wert von a nach folgendem Codeauschnitt?

```
1 int arr[3];
2 arr = (int [3]) {1,2,3};
3 int a = arr[1];
```

- ☐ 2
- ☐ 0
- ☐ Es kommt zu einem Compilerfehler

Quiz: Arrays (3)

Was ist der Wert von a nach folgendem Codeauschnitt?

```
1 int a = ((const int [3]) {1,2,3})[0];
```

- ☐ 1
- ☐ 0
- ☐ Es kommt zu einem Compilerfehler

Quiz: Unterschiede zwischen Arrays und Pointern

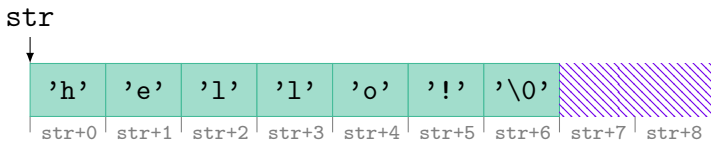
Welche der folgenden Aussagen sind korrekt?

- ☐ Ein Array lässt sich immer wie ein Pointer verwenden.
- ☐ Ein Pointer lässt sich immer wie ein Array verwenden.
- ☐ Arrays können als Funktionsargument by-value übergeben werden.
- ☐ Der `sizeof`-Operator ist sowohl für Arrays als auch für Pointer definiert.

Strings

- Repräsentation als `char`-Array
- Länge implizit: 0-Byte als Terminal

- Üblicherweise: `ASCII`-Zeichensatz bzw. -Kodierung (`man 3 ascii`)



Strings: Definitionsmöglichkeiten

- Array mit explizitem null-Byte:
  ```
  char str[] = {'t', 'u', 'x', '\0'};
  ```
  - mit Hexadezimalwerten (siehe `man 3 ascii`):
    ```
    char str[] = {0x68, 0x75, 0x68, 0x75, 0};
    ```

- Mittels eines String-Literals mit implizitem null-Byte:
  ```
  char str[] = "tux";
  ```
  (äquivalent zur vorigen Variante)

- Über einen Pointer statt einem Array:
  ```
  char* str = "tux";
  ```
  (String hier nicht modifizierbar!)

Quiz: Strings (1)

Was ist der Unterschied zwischen den folgenden Möglichkeiten, einen String zu erstellen?

```
1 char str1[] = "tux";
2 char str2[4] = "tux";
3 char str3[] = {'t', 'u', 'x', '\0'};
4 char str4[4] = {'t', 'u', 'x', '\0'};
```

- ☐ Mindestens eine Möglichkeit ist falsch.
- ☐ Bei den ersten beiden fehlt das terminierende null-Byte.
- ☐ Die ersten beiden Strings können nicht verändert werden.
- ☐ Es gibt keinen Unterschied, das Resultat ist in allen Fällen identisch.

Quiz: Strings (2)

Welchen Wert haben `len` und `size` am Ende dieses Codeabschnittes?

```
1 char str[] = "tux";
2 size_t len = strlen(str);
3 size_t size = sizeof(str);
```

- ☐ `len`: 3, `size`: 3
- ☐ `len`: 3, `size`: 4
- ☐ `len`: 4, `size`: 3
- ☐ `len`: 4, `size`: 4

Quiz: Strings (3)

Was könnte ein mögliches Problem bei folgendem String sein?

```
1 char str[3] = "tux";
```

- [ ] Bei Strings darf es keine explizite Größenangabe für den Array geben.
- [ ] Im resultierenden Array wird das terminierende null-byte fehlen.
- [ ] Es gibt kein Problem.
- [ ] Es kommt zum Compiler-Fehler.

Quiz: Strings (4)

Was könnte an folgendem Codeauschnitt problematisch sein?

```
1 char str[] = "tux";
2 char* str_ptr = str;
3 str_ptr[3] = 'i';
```

- ☐ Nichts, alles in Ordnung.
- ☐ Eine Modifikation eines Strings über einen Pointer ist immer *undefined behavior*.
- ☐ Der Zugriff auf das vierte Element des Arrays ist out of bounds, da der String nur aus drei Zeichen besteht.
- ☐ Das terminierende null-byte wird überschrieben.
- ☐ `str_ptr` zeigt auf die Array-Variable und nicht direkt auf den Array, somit ist der Zugriff semantisch falsch und zudem out of bounds.

Quiz: Strings (5)

Inwiefern könnte folgender Codeauschnitt problematisch sein?

```
1 char* str = "tux";
2 str[0] = 'l';
```

- ☐ Modifikationen von String Literals sind Undefined Behavior.
- ☐ Gar nicht, alles in Ordnung.
- ☐ Es kann zu einem Segmentation Fault kommen.
- ☐ `str` sollte als `const char*` deklariert werden.
- ☐ Auf einem Linux-System ist es nicht möglich, den String "tux" zu verändern.
- ☐ Ein String kann nicht auf diese Weise erstellt werden, es wird ein Array benötigt, kein Pointer.

Quiz: Strings (6)

Welche Aussagen bezüglich des Codes sind korrekt?

```
1 void fn(/* ... */) {
2     char str1[] = "abcd";
3     const char* str2 = "abcd";
4     /* ... */
5 }
```

- ☐ `str2` zeigt auf den Stack.
- ☐ `str2` ermöglicht Speicherplatzoptimierungen.
- ☐ Der Wert von `str1` muss bei jedem Funktionsaufruf neu auf den Stack kopiert werden.
- ☐ `str2` kann auf `str1` zeigen.
- ☐ `str2` ist zu bevorzugen, wenn der String nicht verändert wird.

Quiz: Strings (7)

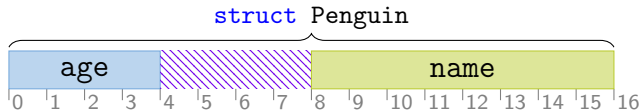
Was ergibt der Ausdruck `"1234" + 1`?

- ☐ Einen Pointer auf den String "1235"
- ☐ Einen Pointer auf den String "12341"
- ☐ Einen Pointer auf den String "234"
- ☐ Den Integer 1235
- ☐ Einen Compiler-Fehler

struct

- Zusammengesetzter Datentyp mit Elementen verschiedenen Typs
- Elemente liegen aufeinanderfolgend (ggf. mit Padding) im Speicher

```
1 struct Penguin {
2   int age;
3   char* name;
4 };
```



- Zugriff direkt oder über einen Pointer

```
1 penguin.age = 0;        // penguin ist vom Typ
2                         // struct Penguin
3 penguin_ptr->age = 0;   // penguin_ptr ist vom Typ
4                         // struct Penguin*
```

struct: Initialisierung

- Initialisierung über "Compound Literal":

```
1 struct Penguin penguin1 = { .age = 0, .name = "Tux" };
2
3 // Implizit: penguin2.name = NULL
4 struct Penguin penguin2 = { .age = 0 };
5
6 // Initialisierung mit festgelegter Reihenfolge
7 struct Penguin penguin3 = { 0, "Tux" };
```

- Initialisierung aller Elemente zu 0:

```
1 struct Penguin penguin4 = { 0 };
```

struct als Parameter

```c
struct Penguin {
    char* name;
    unsigned age;
};
void print_penguin_name1(struct Penguin *penguin) {
    printf("name: %s\n", penguin->name);
}
void print_penguin_name2(struct Penguin penguin) {
    printf("name: %s\n", penguin.name);
}
int main(void) {
    struct Penguin penguin = { "tux", 5 };
    print_penguin_name1(&penguin);
    print_penguin_name2(penguin);
}
```

Quiz: struct (1)

Was ist der Wert (dezimal) von `size` nach folgendem Codeauschnitt? (Wir nehmen an, dass `sizeof(int) == 4` und dass ein `int` ein 4-Byte Alignment erfordert.)

```
1 struct Penguin {
2     char id_bytes[5];
3     int id;
4 };
5 size_t size = sizeof(struct Penguin);
```

- ☐ 9
- ☐ 15
- ☐ 12
- ☐ 16

Quiz: struct (2)

Ist am Ende dieses Codeauschnitts der Wert von `size1` gleich dem von `size2`? (Wir nehmen an, dass `sizeof(int) == 4` und dass ein `int` ein 4-Byte Alignment

```
1  struct Penguin1 {
2      int id;
3      unsigned char age;
4      char color;
5  };
6  struct Penguin2 {
7      unsigned char age;
8      int id;
9      char color;
10 };
11 size_t size1 = sizeof(struct Penguin1);
12 size_t size2 = sizeof(struct Penguin2);
```

☐ Nein

☐ Ja

☐ Es kommt zu einem Compilerfehler

Code zu Quiz: struct (3)

```c
struct Penguin {
    char *name; unsigned age;
};
struct Penguin * create_penguin1(void) {
    struct Penguin penguin = { .name = "tux", .age = 5 };
    return &penguin;
}
struct Penguin create_penguin2(void) {
    struct Penguin penguin = { .name = "tux", .age = 5 };
    return penguin;
}
int main(void) {
    struct Penguin *penguin1 = create_penguin1();
    struct Penguin penguin2 = create_penguin2();
    printf("name1: %s, name2: %s\n", penguin1->name, penguin2.
        name);
}
```

Quiz: struct (3)

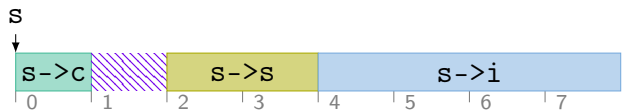
Was ist der Unterschied zwischen den beiden auf der vorigen Folie zu sehenden Ansätzen, ein `struct` als Rückgabewert zu nutzen?

- ☐ Es gibt keinen Unterschied.
- ☐ Variante 1 führt zu einem Segmentation Fault.
- ☐ Variante 2 führt zu einer Compilerwarnung, da ein `struct` nur über einen Pointer zurückgegeben werden kann.
- ☐ Variante 1 führt zu einer Compilerwarnung.

struct vs. union

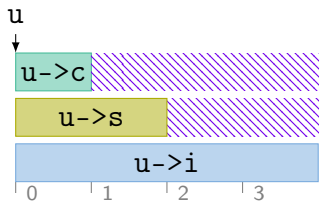
- `struct` speichert immer alle Elemente

```
struct { char c; short s; int i; }* s;
```



- `union` speichert genau ein Element – das zuletzt geschriebene

```
union { char c; short s; int i; }* u;
```



struct mit union

Kombination mit `struct` mit Indikator für Gültigkeit der `union`-Elemente:

```
1  struct Dimension { /* ... */ };
2  struct Shape {
3      int shape_kind; /* 1 = circle, 2 = rect */
4      union {
5          int circle_radius;
6          struct Dimension rect;
7      };
8  };
9
10 // ...
11
12 struct Shape my_circ =
13         { .shape_kind = 1, { .circle_radius = 10 }};
```

union als Parameter

```
1 void f(union Number num) {
2     // Zugriff auf das Attribut ‘a‘ mit ‘num.a‘ ...
3 }
4
5 void g(union Number *num) {
6     // Zugriff auf das Attribut ‘a‘ mit ‘num->a‘ ...
7 }
```

- Zugriff auf die Member einer `union` syntaktisch analog zu `struct`

Quiz: union (1)

Was ist der Wert (dezimal) von `value` nach folgendem Codeauschnitt? (Wir nehmen an, dass `sizeof(int) == 4` und `CHAR_BIT == 8`.)

```
1 union Number {
2     char a;
3     int  b;
4 };
5 union Number num = {.b = 0x10000};
6 int value = num.a;
```

- ☐ 0
- ☐ 16
- ☐ Es kommt zu einem Compilerfehler.
- ☐ Das Ergebnis ist nicht spezifiziert.

Quiz: union (2)

Was ist der Wert (dezimal) von `size` nach folgendem Codeauschnitt? (Wir nehmen an, dass `sizeof(int) == 4` und dass ein `int` ein 4-Byte Alignment erfordert.)

```
1 union Penguin {
2     char id_bytes[5];
3     int id;
4 };
5 size_t size = sizeof(union Penguin);
```

- ☐ 8
- ☐ 5
- ☐ 9

Quiz: union (3)

Was ist der Wert (dezimal) von `size` nach folgendem Codeauschnitt?

```
1 union Penguin {
2     char id_short[5];
3     char id_long[7];
4 };
5 size_t size = sizeof(union Penguin);
```

- ☐ 5
- ☐ 7
- ☐ 8
- ☐ 35

Verschachtelte Datentypen

Mehrdimensionale Arrays - Iterationsreihenfolge

```
unsigned matrix[3][4] = {
    { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 }
};

for (size_t i = 0; i < sizeof(matrix) / sizeof(*matrix); i++) {
    for (size_t j = 0;
         j < sizeof(matrix[i]) / sizeof(*matrix[i]);
         j++) {
        printf("%u ", matrix[i][j]);
    }
}

printf("\n");
```

Verschachtelte Datentypen

Mehrdimensionale Arrays – implizite Größe

Auslassen der expliziten Größenangabe der “obersten Ebene”:

```
1 unsigned matrix[][4] = {
2     { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 }
3 };
```

Quiz: Mehrdimensionale Arrays (1)

Was ist der Wert von `size` nach folgendem Codeauschnitt?

```
1 char matrix[8][9];
2 size_t size = sizeof(matrix);
```

- ☐ 9
- ☐ 72
- ☐ 8
- ☐ Es kommt zu einem Compilerfehler.

Quiz: Mehrdimensionale Arrays (2)

Was ist der Wert von `size` nach folgendem Codeauschnitt?

```
1 char matrix[5][];
2 size_t size = sizeof(matrix / *matrix);
```

- ☐ 20
- ☐ 5
- ☐ 4
- ☐ Es kommt zu einem Compilerfehler.

Code zu Quiz: Mehrdimensionale Arrays (3)

```
1  // Loop 1
2  for (size_t i = 0; i < sizeof(matrix) / sizeof(*matrix); i++) {
3      for (size_t j = 0;
4           j < sizeof(matrix[i]) / sizeof(*matrix[i]);
5           j++) {
6          matrix[i][j] <<= 1;
7      }
8  }
9  // Loop 2
10 for (size_t i = 0; i < sizeof(matrix) / sizeof(*matrix); i++) {
11     for (size_t j = 0;
12          j < sizeof(matrix[i]) / sizeof(*matrix[i]);
13          j++) {
14             matrix[j][i] <<= 1;
15     }
16 }
```

Quiz: Mehrdimensionale Arrays (3)

Welcher Unterschied folgt aus der unterschiedlichen Iterationsreihenfolge über das zweidimensionale Array `matrix` in den beiden `for`-Loops im folgenden Codeauschnitt?

- ☐ `Loop 2` führt zu einem Segmentation Fault.
- ☐ Es gibt keinen funktionalen Unterschied.
- ☐ Es kommt zu einem Compilerfehler.
- ☐ `Loop 1` wird im Allgemeinen etwas schneller sein.

Quiz: Verschachtelte Datenstrukturen (1)

Was könnte an folgendem Codeauschnitt problematisch sein?

```
1 struct Penguin {
2     struct {
3         unsigned height;
4         unsigned age;
5         unsigned id;
6     };
7     char id[256];
8 };
```

- ☐ Ein verschachteltes `struct` muss immer benannt sein.
- ☐ `unsigned` alleine ist kein valider Datentyp.
- ☐ Innerhalb eines `structs` darf ein weiteres `struct` nicht das erste Element sein.
- ☐ Es kommt zu einem Compilerfehler.

Code zu Quiz: Verschachtelte Datenstrukturen (2)

```
1  struct Penguin {
2      char color;
3      long id;
4  };
5  struct PenguinColony {
6      char color[256];
7      long ids[256];
8  };
9  struct Penguin penguin_colony1[256];    // Array of Structs
10 struct PenguinColony penguin_colony2;   // Struct of Arrays
```

Quiz: Verschachtelte Datenstrukturen (2)

Was könnte im Allgemeinen ein wesentlicher Unterschied sein zwischen den auf der vorigen Folie zu sehenden Möglichkeiten, mit mehreren gleichartigen `structs` umzugehen? Sollte lieber ein separates `struct` mit Arrays der einzelnen Member gewählt werden oder sollte einfach ein Array des ursprünglichen `structs` `Penguin` gewählt werden?

- ☐ Im Allgemeinen gibt es keinen Unterschied.
- ☐ Je nach Zugriffsmuster kann es einen Performance-Unterschied geben.
- ☐ Ein Struct of Arrays kann Speichereffizienter sein, da Padding innerhalb der `structs` entfällt.

Mini-"OOP" in C

```c
struct Penguin {
    char *name;
    int age;
    long position;
    void (*walk)(struct Penguin* penguin);
};
void walk(struct Penguin*  penguin) {
    penguin->position++;
}
int main(void) {
    struct Penguin penguin = {
        .name = "tux", .age = 5, .position = 10, .walk = walk
    };
    penguin.walk(&penguin);
    printf("%ld\n", penguin.position);  // -> 11
}
```

Quiz: Mini-"OOP" in C

Im vorigen Beispiel wurde der `walk`-Funktion der `Penguin` per Pointer übergeben. Wie unpraktisch! Man kann ein `struct` doch auch in diesem Beispiel direkt als Parameter spezifizieren, oder?

- ☐ Ja, das wäre auch besserer Code-Stil
- ☐ Nein, denn dann wird in `walk` nur eine Kopie des ursprünglichen `structs` verändert.
- ☐ Nein, man kann ein `struct` nicht direkt übergeben.