This document summarizes key concepts and examples related to C Structures, as presented in "COMP 206 Lesson 16 Struct – Software Systems" from McGill Slides by Vybihal & Bérubé-Vallières © 2023.

C Structures (COMP 206 - Lesson 16)

Source: McGill Slides – Vybihal & Bérubé-Vallières © 2023
Textbook Reference: Section 3.0, Example 7
Topic: Struct and Union Notes

---

1. Introduction to Structures

- **Definition:** Structures (`struct`) are a collection of items that may have different data types.
- **Elements:** Their elements are called **members**.
- **Access:** Unlike arrays, members are accessed by their **name**.

---

2. Declaring and Accessing Structures (Anonymous)

Initial Declaration (Global Variable)

Structures can be declared without a tag, making them "anonymous."

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

struct {            // Anonymous struct
    char name[30];
    int age;
    float salary;
} a;                // Declares 'a' as a g

int main(void) {
    a.age = 25;                          //
    strcpy(a.name, "John");              //
    printf("%s is %d years old\n", a.nam
}
```

- **Accessing Members:** Use the **dot operator (`.`)** (e.g., `a.age`).
- **Variable Declaration:** Follows the usual form: `type variable_name;`
- **Important:** Note the semicolon after the closing brace `}` .

Declaring Multiple Variables (Anonymous)

You can declare multiple variables of the same anonymous struct type at once:

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

struct {
    char name[30];
    int age;
    float salary;
} a, b; // Declares 'a' and 'b' as global variables

int main(void) {
    a.age = 25;
    strcpy(a.name, "John");
    b.age = 30;
    strcpy(b.name, "Bob");
    printf("%s is %d years old, %s is %d years old\n", a.name, a.age, b.name, b.age);
}
```

Problem with Anonymous Local Structures:
Declaring the same anonymous structure multiple times locally is cumbersome and repetitive, as each declaration creates a distinct type.

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

int main(void) {
    struct { // Anonymous struct type 1
        char name[30];
        int age;
        float salary;
    } a;
    /* ... */
    struct { // Anonymous struct type 2
        char name[30];
        int age;
        float salary;
    } b;
}
```

This is generally not recommended for repeated use.

---

3. Structure Tags (Named Structures)

- **Purpose:** Tags solve the problem of repetitive and distinct anonymous structure declarations, especially for local variables.
- **Declaration:** A tag gives a name to the structure definition.

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

struct PERSON {       // 'PERSON' is the tag
    char name[30];
    int age;
    float salary;
};                    // Semicolon here, even if no variables declared

int main(void) {
    struct PERSON a;  // Declares 'a' of type 'struct PERSON'
    /* ... */
    struct PERSON b;  // Declares 'b' of type 'struct PERSON'
}
```

The tag replaces the need to redefine the members.

Global and Multiple Variable Declarations with Tags:
You can still declare global variables or multiple variables at once when using a tag.

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

struct PERSON {
    char name[30];
    int age;
    float salary;
} d; // 'd' is a global variable of type

int main(void) {
    struct PERSON a, b; // 'a' and 'b' a
    /* ... */
    struct PERSON c;    // 'c' is anothe
}
```

4. General Syntax of `struct`

```
struct OPTIONAL_TAG {
    MEMBERS;
};
```

- `OPTIONAL_TAG`: A user-defined name for the structure. If not present, the struct is anonymous.
- `MEMBERS`: A list of semicolon-separated variable declarations, e.g.:

  ```
  TYPE1 VAR1;
  TYPE2 VAR2;
  // etc.
  ```

- **Important Restriction:** You cannot initialize members from inside the `struct` definition (e.g., `int age = 0;` is not allowed here).

---

5. The Dot Operator (`.`)

Used to access fields (members) within a structure.

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

struct PERSON {
    char name[30];
    int age;
    float salary;
};

int main(void) {
    struct PERSON a;
    scanf("%s", a.name);          // Acce
    scanf("%d", &(a.age));        // Acce
    a.salary = 50.25;             // Assi
    printf("%s %d %f", a.name, a.age, a.
}
```

6. Struct Assignment

- **Behavior:** Unlike arrays, you can use the assignment operator (`=`) with structs.
- **Compatibility:** This works if the structs have compatible types (declared with the same tag, or if anonymous, declared at the same time).
- **Mechanism:** The assignment copies every member's value one by one (e.g., `b.x = a.x; b.y = a.y;`).
- **Arrays within Structs:** This even works with arrays that are members of structs!

```
#include <stdio.h>

struct COORDINATE {
    int x;
    int y;
};

int main(void) {
    struct COORDINATE a, b;
    a.x = 5;
    a.y = 7;
    b = a; // Copies all members from 'a
    printf("The coordinate of b is (%d,

    // Example with array inside struct
    struct { int arr[5]; } c, d;
    c.arr[0] = 100;
    d = c;
    printf("d.arr[0] is %d\n", d.arr[0])
}
```

---

7. Nested Structures

- **Concept:** Structures can be used as members within other structures.
- **Access:** Use chained dot operators (e.g., `parent.child.member`).
- **Functions:** Structures can be passed as arguments to functions.

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

struct NAME {
    char first[30];
    char last [30];
};

struct PERSON {
    struct NAME name; // Nested struct
    int age;
    float salary;
};

void print_name(struct NAME n) { // Stru
    printf("%s %s\n", n.first, n.last);
}

int main(void) {
    struct PERSON a;
    strcpy(a.name.first, "John"); // Cha
    strcpy(a.name.last, "Smith"); // Cha
    print_name(a.name);
}
```

8. `typedef` with Structures

- **Purpose:** `typedef` is often used with structs to create an alias (a new name) for the structure type, avoiding the need to use the `struct` keyword repeatedly.
- **Syntax:** `typedef type new_name;`

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

typedef struct {
    char name[30];
    int age;
    float salary;
} Person; // 'Person' is now an alias fo

int main(void) {
    Person a, b; // No 'struct' keyword
    /* ... */
    Person c;
}
```

- `Person` in this context is the new type name, not a variable declaration.

---

9. Array of Structs

You can create arrays where each element is a structure.

```
#include <stdio.h>

struct PERSON {
    char name[30];
    int age;
    float salary;
};

int main() {
    struct PERSON people[100]; // Array

    for(int x = 0; x < 100; x++) {
        printf("Enter name for person %d
        scanf("%s", people[x].name); //
        printf("Enter age for person %d:
        scanf("%d", &(people[x].age));
        printf("Enter salary for person
        scanf("%f", &(people[x].salary))

        if (people[x].age == 20) {
            printf("Hey you are 20!!\n")
        }
    }
}
```

---

10. `sizeof` and Structs (Memory Alignment / Padding)

- **Memory Placement:** In memory, members of a struct are generally placed in sequence.
- **Padding:** However, compilers often require addresses of certain items to be on specific multiples (e.g., integers on multiples of 4). To achieve this, the compiler may insert **unused bytes (padding)** between elements.
- **Impact on `sizeof`:** Because of padding, the `sizeof` a struct might be larger than the simple sum of the sizes of its individual members.

```
struct PERSON {
    char name[30]; // 30 bytes
    int age;       // 4 bytes (example)
    float salary;  // 4 bytes (example)
};
// Expected sum: 30 + 4 + 4 = 38 bytes
// Actual sizeof(struct PERSON) might be, for example, 40 or 44 bytes due to padding.
```

The layout in memory might look like: `name | padding | age | padding | salary | padding (at end of struct)`

---

11. Recap

- **Syntax:**
  - `struct OPTIONAL_TAG { MEMBERS; } OPTIONAL_VARIABLES;`
  - `typedef struct { MEMBERS; } NEW_TYPE_NAME;`
- **Function Interaction:** You can pass structs by value or by reference (using pointers) as arguments to functions, and you can also return them from functions.
- **Convenience:** Using structs makes it much more convenient to group and move related data, including arrays, as a single logical unit.
- **Size:** The size of a struct (`sizeof`) is not simply the sum of the sizes of its fields, as padding may be added for memory alignment purposes.

---

12. Practice Problem

Write a C program that does the following:

1. Define a `struct Coord` type representing a coordinate in a 2D space (X and Y).
2. Implement a function `CoordArray* read_coords(int *capacity)` that:
   - Allocates a dynamic array named `CoordArray` of `struct Coord` type with a user-chosen capacity.
   - Reads coordinate inputs (X and Y) from the user and stores them in the dynamically allocated `CoordArray`.
   - Returns a pointer to the allocated `CoordArray`.
3. Implement a function `Coord centroid(Coord *arr, int capacity)` that computes the centroid (average X and Y) of all coordinates entered.
4. In the `main` function, output the computed centroid to the user.