C Structures (Struct) - COMP 206 Lesson 16

Course: Unix Bash C GNU Systems COMP 206 – Software Systems
Lesson: Struct – Software Systems
References:

- Textbook: Section 3.0, Example 7
- Source: McGill Slides – Vybihal & Bérubé-Vallières © 2023

1. Introduction to Structures

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

2. Declaring Structures

2.1 Anonymous Structures (Examples 1 & 2)

An anonymous structure is declared without a tag. Variables of this type must be declared directly after the structure definition.

Declaration as a Global Variable:

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

struct { // This defines an anonymous struct type
    char name[30];
    int age;
    float salary;
} a; // 'a' is a global variable of this anonymous st

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

- **Note:** The declaration follows the usual form: `type` (the anonymous `struct {...}`) followed by the `variable name` (`a`).
- **In Reality:** You would most likely use a structure tag, as seen later in this lesson.

Declaring Multiple Variables:

Multiple variables of the same anonymous type can be declared at once, similar to de multiple variables of a primitive type.

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

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

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

Note: Declarations can be of the form "`ty var_name1, var_name2, ...`".

2.2 Problem with Anonymous Local Structures (Example 3)

Declaring the same anonymous structure loca different places leads to cumbersome and rep code, as each instance effectively defines a n distinct type.

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

int main(void) {
    struct { // First local anonymous struct
        char name[30];
        int age;
        float salary;
    } a;
    /* ... */
    struct { // Second local anonymous struct (duplicate definition)
        char name[30];
        int age;
        float salary;
    } b;
}
```

- **Issue:** Each `struct { ... }` block creates a *new*, unique anonymous type. Variables `a` and `b` in this example are of incompatible types despite having identical member definitions.
- **Solution:** A better way is to use structure tags.

2.3 Structures with Tags (Examples 4 & 5)

Using a tag allows you to define a structure type once and then declare variables of that type anywhere, avoiding repetition and ensuring type compatibility.

Defining a Tagged Structure:

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

struct PERSON { // 'PERSON' is the
    char name[30];
    int age;
    float salary;
}; // Notice the semi-colon even th
int main(void) {
    struct PERSON a; // Declaring a
    /* ... */
    struct PERSON b; // Declaring a
}
```

Benefit: The tag `PERSON` replaces the ne to redefine the members repeatedly.

Combining Tags with Variable Declaratio

You can still declare variables (global or loc single or multiple) simultaneously with the tagged structure definition.

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

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

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

2.4 General Syntax of `struct` (Slide 9)

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

- `OPTIONAL_TAG`: A user-defined tag (e.g., `PERSON`). If not present, the struct is anonymous.
- `MEMBERS`: A list of semi-colon separated variable declarations defining the struct's elements.

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

> **Important Note:** You cannot initialize m
> inside the struct definition (e.g., you can
> `age = 0;`).

3. Accessing Structure Members: The Dot Operator

The dot operator (`.`) is used to access individual fields (members) within a structure variable.

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

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

int main(void) {
    struct PERSON a;

    // Reading input into struc
    printf("Enter name: ");
    scanf("%s", a.name);
    printf("Enter age: ");
    scanf("%d", &(a.age));

    // Assigning a value
    a.salary = 50.25;

    // Printing struct members
    printf("%s %d %f\n", a.name

    return 0;
}
```

4. Structure Assignment (Examples 13 & 14)

Unlike arrays, you can use the assignment operator (`=`) with entire structs, provided they 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;`).
- **Compatibility:** This works even if structs contain arrays as members; the array contents will be copied.

```
#include <stdio.h>

struct COORDINATE {
    int x;
    int y;
};

int main(void) {
    struct COORDINATE a, b;
    a.x = 5;
    a.y = 7;

    b = a; // Struct assignment

    printf("The coordinate of b

    // Example with arrays insi
    struct { int arr[5]; } c, d
    c.arr[0] = 5;
    d = c; // Copies the entire
    printf("d.arr[0] is %d\n",

    return 0;
}
```

5. Nested Structures (Example 15)

Structures can be used as members within other structures, allowing for more complex data organization.

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

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

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

// Structures can be passed as
void print_name(struct NAME n)
    printf("%s %s\n", n.first,
}

int main(void) {
    struct PERSON a;

    // Accessing members of nes
    strcpy(a.name.first, "John"
    strcpy(a.name.last, "Smith"

    print_name(a.name); // Pass

    return 0;
}
```

6. Typedef and Structures (Example 16)

`typedef` is often used with structures to create a new alias (type name) for the struct type, allowing you to declare variables without needing the `struct` keyword.

- **Syntax:** `typedef type new_name;`
- **Benefit:** Makes code cleaner and easier to read.

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

// Defines a new type 'Person'
typedef struct {
    char name[30];
    int age;
    float salary;
} Person; // 'Person' is the ne

int main(void) {
    Person a, b; // Declaring v
    /* ... */
    Person c;

    strcpy(a.name, "Alice");
    a.age = 30;
    a.salary = 60000.0f;
    printf("%s is %d years old

    return 0;
}
```

7. Array of Structs (Example 17)

You can declare arrays where each element is a structure. This allows managing collections of related data.

```c
#include <stdio.h>

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

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

    for(int x = 0; x < 100; x++
        printf("Enter name for
        scanf("%s", people[x].n
        printf("Enter age for p
        scanf("%d", &(people[x]
        printf("Enter salary fo
        scanf("%f", &(people[x]

        if (people[x].age == 20
            printf("Hey you are

        // For demonstration, b
        if (x == 1) break;
    }

    printf("\nFirst two persons
    printf("1. Name: %s, Age: %
    printf("2. Name: %s, Age: %

    return 0;
}
```

8. Sizeof and Structures (Memory Alignment) (Example 18)

- **Memory Placement:** In memory, members of a struct are conceptually placed in sequence.
  `struct PERSON { char name[30]; int age; float salary; };`
  Visual representation in memory: `[name (30 bytes)] [age (4 bytes)] [salary (4 bytes)]` (assuming typical sizes)
- **Memory Alignment/Padding:** Computers often require addresses of certain items (like integers) to be on specific memory multiples (e.g., an `int` might always start on an address divisible by 4).
  - To satisfy these alignment requirements, the compiler may insert unused bytes (called **padding**) between members.
- **Consequence:** Because of padding, the `sizeof` a struct might be larger than the simple sum of the sizes of its individual members.

9. Recap and Key Takeaways

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

10. Practice Problem

Write a C program that does the following:

1. **Define** `struct Coord`**:** Create a structure type named `Coord` representing a coordinate in a 2D space (with `x` and `y` integer members).
2. **Implement** `read_coords(int *capacity)` **function:**
   - Dynamically allocate an array named `CoordArray` of `struct Coord` type.
   - The capacity of this array should be determined by user input (passed via `capacity` pointer).
   - Read coordinate inputs (x, y pairs) from the user and store them in the dynamically allocated `CoordArray`.
   - Return a pointer to the allocated `CoordArray`.
3. **Implement** `Coord centroid(Coord *arr, int capacity)` **function:**
   - Compute the centroid (average X and average Y) of all coordinates stored in the `arr`.
   - Return a `Coord` struct representing the centroid.
4. `main` **function:**
   - Call `read_coords` to get user coordinates.
   - Call `centroid` to calculate the centroid.
   - Output the computed centroid to the user.
   - Remember to free dynamically allocated memory.