Trainer: Asif Nafees

Structures

A structure in C is a user-defined data type that allows grouping different types of data items under a single name. It is similar to class in other languages. Structures are used when we want to store multiple related variables of different data types under one name.

Key Points about Structures
Syntax: Structures are defined using the struct keyword.
Multiple Data Types: Unlike arrays, structures can contain different data types (int, float, char, etc.).
Memory Allocation: The size of a structure is the sum of the sizes of its members, and memory is allocated contiguously.

Defining a Structure
```
struct Student {
   char name[50];
   int age;
   float marks;
};
```
struct is the keyword used to define a structure.

Declaring Structure Variables
You can declare structure variables in two ways:
1. Separate Declaration:
```
struct Student s1, s2;
```
2. Direct Declaration:
```
struct {
   char name[50];
   int age;
   float marks;
} s1, s2;
```

Accessing Structure Members
The members of a structure can be accessed using the dot (.) operator.
```
#include <stdio.h>
struct Student {
   char name[50];
   int age;
   float marks;
};
int main() {
   struct Student s1;
   // Input
```

```
    printf("Enter name: ");
    scanf("%s", s1.name);
    printf("Enter age: ");
    scanf("%d", &s1.age);
    printf("Enter marks: ");
    scanf("%f", &s1.marks);
    // Output
    printf("\nStudent Details:\n");
    printf("Name: %s\n", s1.name);
    printf("Age: %d\n", s1.age);
    printf("Marks: %.2f\n", s1.marks);
    return 0;
}
```

Explanation:
s1.name, s1.age, and s1.marks are used to access the members of the structure.
The dot (.) operator is used to access individual members.

Array of Structures
You can create an array of structures to store information about multiple entities.

```
#include <stdio.h>
struct Student {
    char name[50];
    int age;
    float marks;
};
int main() {
    struct Student s[3];
    for (int i = 0; i < 3; i++) {
        printf("Enter details for student %d\n", i + 1);
        printf("Enter name: ");
        scanf("%s", s[i].name);
        printf("Enter age: ");
        scanf("%d", &s[i].age);
        printf("Enter marks: ");
        scanf("%f", &s[i].marks);
    }
    printf("\nDisplaying Student Information:\n");
    for (int i = 0; i < 3; i++) {
        printf("Name: %s, Age: %d, Marks: %.2f\n", s[i].name, s[i].age, s[i].marks);
    }
    return 0;
}
```

Explanation:
s[i].name, s[i].age, and s[i].marks are used to access the members of each student.

Nested Structures

A structure can contain another structure as its member.

```
#include <stdio.h>
struct Address {
    char city[50];
    int pincode;
};
struct Student {
    char name[50];
    int age;
    struct Address addr;
};

int main() {
    struct Student s1;
    printf("Enter name: ");
    scanf("%s", s1.name);
    printf("Enter age: ");
    scanf("%d", &s1.age);
    printf("Enter city: ");
    scanf("%s", s1.addr.city);
    printf("Enter pincode: ");
    scanf("%d", &s1.addr.pincode);
    printf("\nStudent Details:\n");
    printf("Name: %s\n", s1.name);
    printf("Age: %d\n", s1.age);
    printf("City: %s\n", s1.addr.city);
    printf("Pincode: %d\n", s1.addr.pincode);
    return 0;
}
```

Explanation:
s1.addr.city and s1.addr.pincode access the members of the nested Address structure.

Pointers to Structures

You can use pointers to structures to access members using the arrow (->) operator.

```
#include <stdio.h>
struct Student {
    char name[50];
    int age;
    float marks;
};
int main() {
    struct Student s1 = {"John", 20, 85.5};
    struct Student *ptr = &s1;
    printf("Name: %s\n", ptr->name);
    printf("Age: %d\n", ptr->age);
    printf("Marks: %.2f\n", ptr->marks);
    return 0;
}
```

Explanation:
ptr->name is equivalent to (*ptr).name.
The arrow (->) operator is used with pointers to structures.

Practice Questions
1. Define a structure for a Book with members: title, author, and price. Write a program to input and display book details.
2. Create a program using an array of structures to store and display information of 5 employees.

Unions

A union in C is a user-defined data type that, like a structure, allows grouping different types of variables under a single name. However, unions differ from structures in terms of memory usage. In a union, all members share the same memory location, meaning only one member can hold a value at a time.

Key Points About Unions
1. Memory Sharing: The size of a union is equal to the size of its largest member. This is because all members share the same memory location.
2. Efficient Memory Usage: Unions are useful when you need to store different types of data but never use them at the same time, saving memory.
3. Mutual Exclusivity: Only one member of a union can be accessed at a time; if you assign a value to one member, the previous value stored in another member is overwritten.

Defining a Union
The syntax for defining a union is similar to that of a structure.

```
union Data {
    int i;
    float f;
    char str[20];
};
```

union is the keyword used to define a union.
i, f, and str are the members of the union.

Declaring Union Variables
You can declare union variables in two ways:
1. Separate Declaration:

```
union Data data1, data2;
```

2. Direct Declaration:

```
union {
    int i;
    float f;
    char str[20];
} data;
```

Accessing Union Members
Union members are accessed using the dot (.) operator, similar to structures.

```
#include <stdio.h>
union Data {
    int i;
    float f;
    char str[20];
};
int main() {
    union Data data;
    data.i = 10;
    printf("Integer: %d\n", data.i);
    data.f = 220.5;
    printf("Float: %.2f\n", data.f);
    // Assigning a string
    strcpy(data.str, "Hello World");
    printf("String: %s\n", data.str);
    // Accessing the integer again
    printf("Integer after string assignment: %d\n", data.i);
    return 0;
}
```

Output:

```
Integer: 10
Float: 220.50
String: Hello World
Integer after string assignment: Garbage Value
```

Explanation:
The value of data.i becomes undefined after assigning a value to data.f and data.str because they all share the same memory location.

Memory Allocation in Unions
The size of a union is determined by the size of its largest member.

```
#include <stdio.h>
union Data {
    int i;        // 4 bytes
    float f;      // 4 bytes
    char str[20]; // 20 bytes
};
int main() {
    printf("Size of union: %lu bytes\n", sizeof(union Data));
    return 0;
}
```

Output:

```
Size of union: 20 bytes
```

Explanation:
The size of the union is 20 bytes, equal to the size of the largest member (str).

Difference Between Structure and Union
Example:
Using Union for Efficient Memory Usage
Unions are useful in scenarios where different types of data are used but only one at a time, such as handling a network packet where different fields are used based on the packet type.

```
#include <stdio.h>
union Packet {
    int intData;
    float floatData;
    char charData;
};
int main() {
    union Packet packet;
    packet.intData = 100;
    printf("Integer data: %d\n", packet.intData);
    packet.floatData = 99.9;
    printf("Float data: %.2f\n", packet.floatData);
    packet.charData = 'A';
    printf("Character data: %c\n", packet.charData);
    return 0;
}
```

Output:

```
Integer data: 100
Float data: 99.90
Character data: A
```

Explanation:
Here, the union allows us to store integer, float, or character data in the same memory location.

Pointers to Unions
Just like structures, we can use pointers to access union members using the arrow (->) operator.

```
#include <stdio.h>
union Data {
   int i;
   float f;
};
int main() {
   union Data data;
   union Data *ptr = &data;
   ptr->i = 5;
   printf("Integer: %d\n", ptr->i);
   ptr->f = 5.5;
   printf("Float: %.2f\n", ptr->f);
   return 0;
}
```

Output:

```
Integer: 5
Float: 5.50
```

Common Mistakes with Unions
1. Accessing Multiple Members: Accessing multiple members of a union at the same time leads to undefined behavior since they share the same memory.
2. Misinterpreting Data: Assigning a value to one member and reading from another member may result in garbage or unexpected values.
3. Incorrect Memory Size: Remember that the size of a union is determined by its largest member, not the sum of its members.

Use Cases of Unions
1. Data Type Interpretation: Unions are often used to interpret a memory location in multiple ways (e.g., as an integer or a float).

2. Efficient Memory Usage: Unions save memory when different types of data are used at different times (e.g., in embedded systems).
3. Variant Data Structures: Unions are useful in creating variant data structures that hold different types of data based on conditions (e.g., a message packet with different data formats).

Practice Questions
1. Define a union to store a value that could be an integer, a float, or a character. Write a program to input and display values for each type.
2. Write a program that demonstrates the difference between structures and unions with examples.
3. Create a union that can store either a short int, a long int, or a double. Print the size of the union and its members.