CSE 06131223 ♦ CSE 06131224

Structured Programming

Lecture 19

Structures and Unions (3)

Prepared by

Md. Mijanur Rahman, Prof. Dr.
Dept. of Computer Science and Engineering
Jatiya Kabi Kazi Nazrul Islam University, Bangladesh
www.mijanrahman.com

Contents

Structures and Unions in C

Structure Within Structure

- In C, "structure within structure" typically refers to the concept of nesting structures, also known as nested structures or structures within structures. This involves defining one structure (let's call it the outer structure) that contains another structure (the inner structure) as one of its members.
- Let us consider the following structure defined to store a person’s information:

```
struct Person {
   char name[50];
   int age;
   int birth_day;
   int birth _month;
   int birth_year;
};
```

- This structure defines name, age, and three kinds of birthday information: day, month, and year. We can group all the items related to the birthday together and declare then under a substructure.

Structure Within Structure

- We can group all the items related to the birthday together and declare then under a substructure as follows:

```
struct Person {
    char name[50];
    int age;
   struct {
       int day;
       int month;
       int year;
   }birthday;
} person1;
```

Structure Within Structure

- We can also declare inner and outer structures as follows:

(1) Define inner structure:

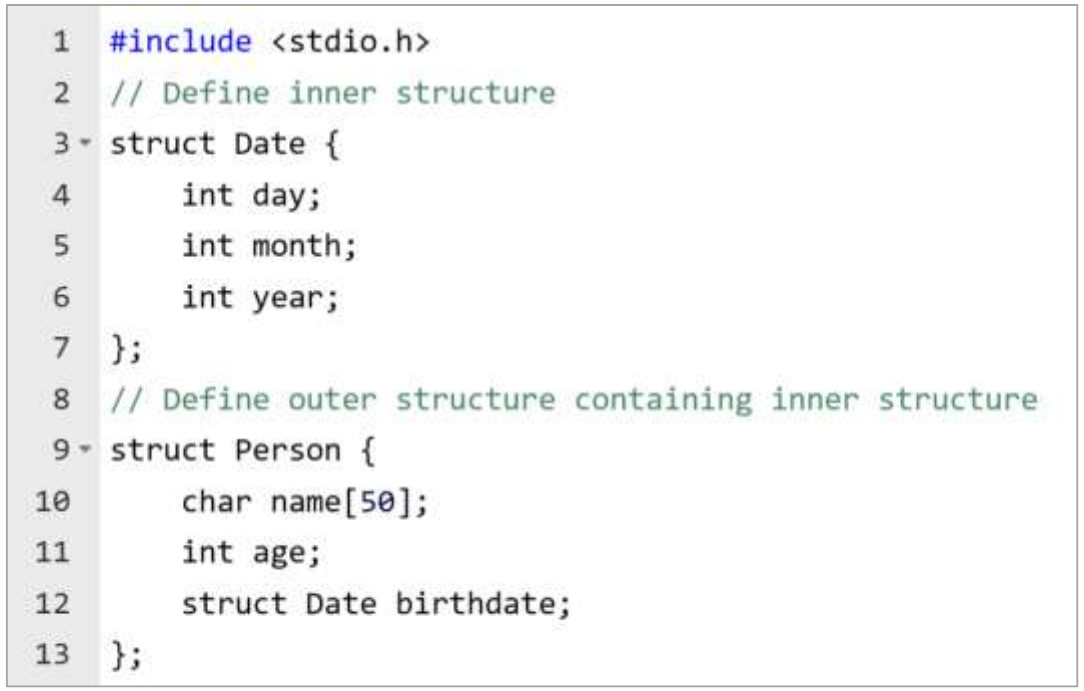
```
struct Date {
        int day;
        int month;
        int year;
};
```

(2) Define outer structure containing inner structure:

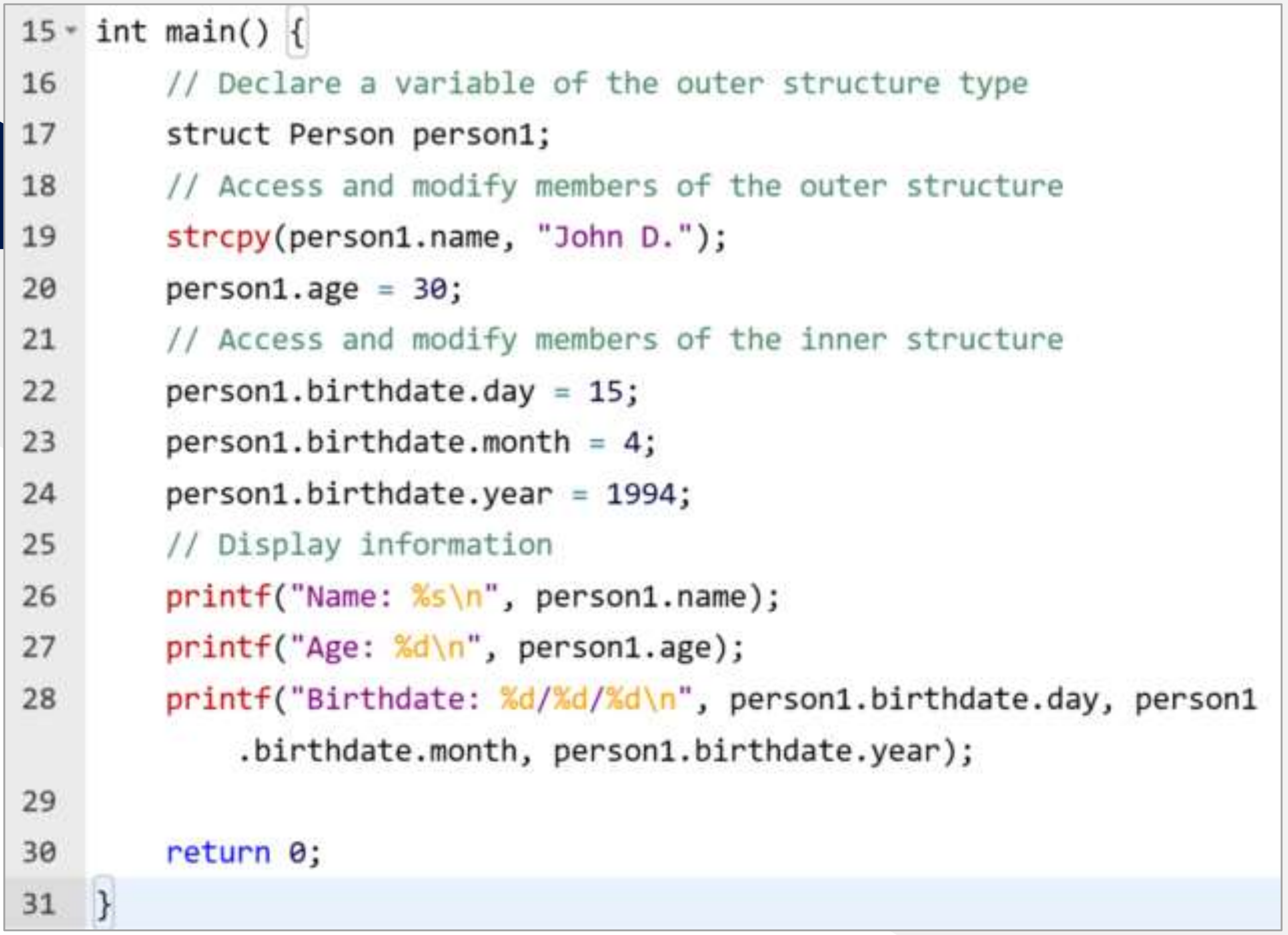
```
struct Person {
    char name[50];
    int age;
   struct Date birthday;
};
```

Structure Within Structure

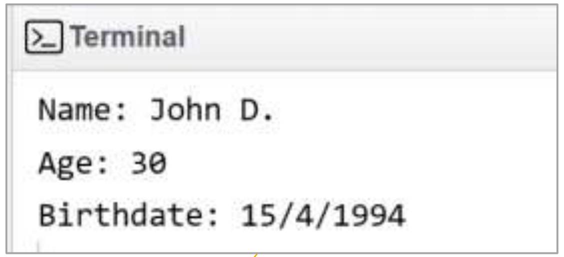
- Here's an example to illustrate this concept:



```c
#include <stdio.h>
// Define inner structure
struct Date {
    int day;
    int month;
    int year;
};
// Define outer structure containing inner structure
struct Person {
    char name[50];
    int age;
    struct Date birthdate;
};
```



```c
int main() {
    // Declare a variable of the outer structure type
    struct Person person1;
    // Access and modify members of the outer structure
    strcpy(person1.name, "John D.");
    person1.age = 30;
    // Access and modify members of the inner structure
    person1.birthdate.day = 15;
    person1.birthdate.month = 4;
    person1.birthdate.year = 1994;
    // Display information
    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    printf("Birthdate: %d/%d/%d\n", person1.birthdate.day, person1.birthdate.month, person1.birthdate.year);

    return 0;
}
```

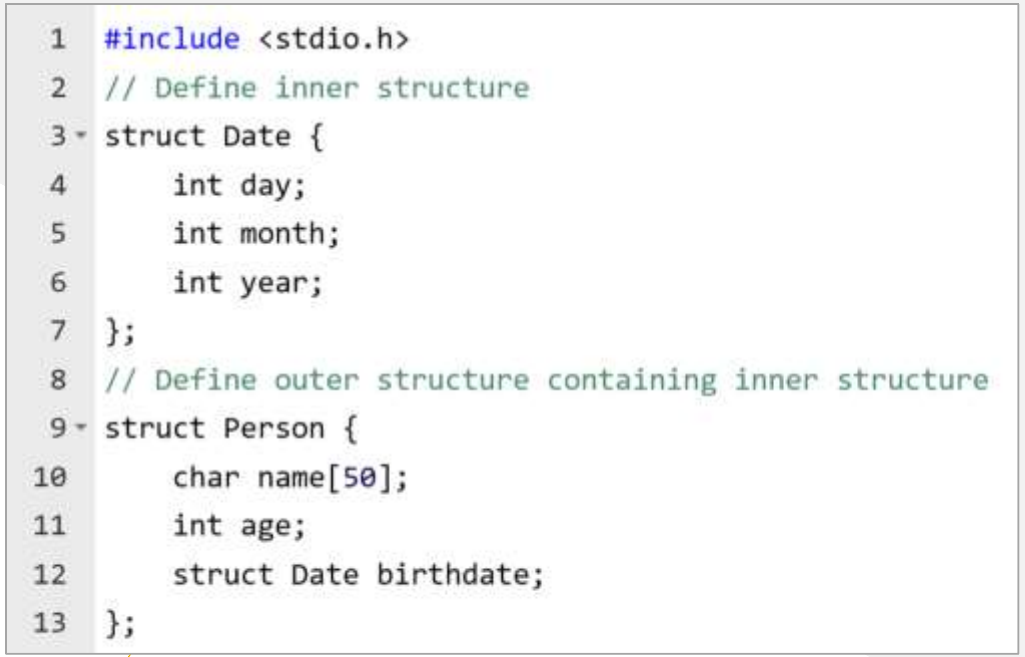


Terminal

```
Name: John D.
Age: 30
Birthdate: 15/4/1994
```

Structure Within Structure

- In this example, we have two structures: struct Date and struct Person.
- The **struct Date** represents a date with day, month, and year fields, while **struct Person** represents a person with a name, age, and birthdate.
- The **struct Person** structure contains an instance of the **struct Date** structure (birthdate) as one of its members.



```c
#include <stdio.h>
// Define inner structure
struct Date {
    int day;
    int month;
    int year;
};
// Define outer structure containing inner structure
struct Person {
    char name[50];
    int age;
    struct Date birthdate;
};
```

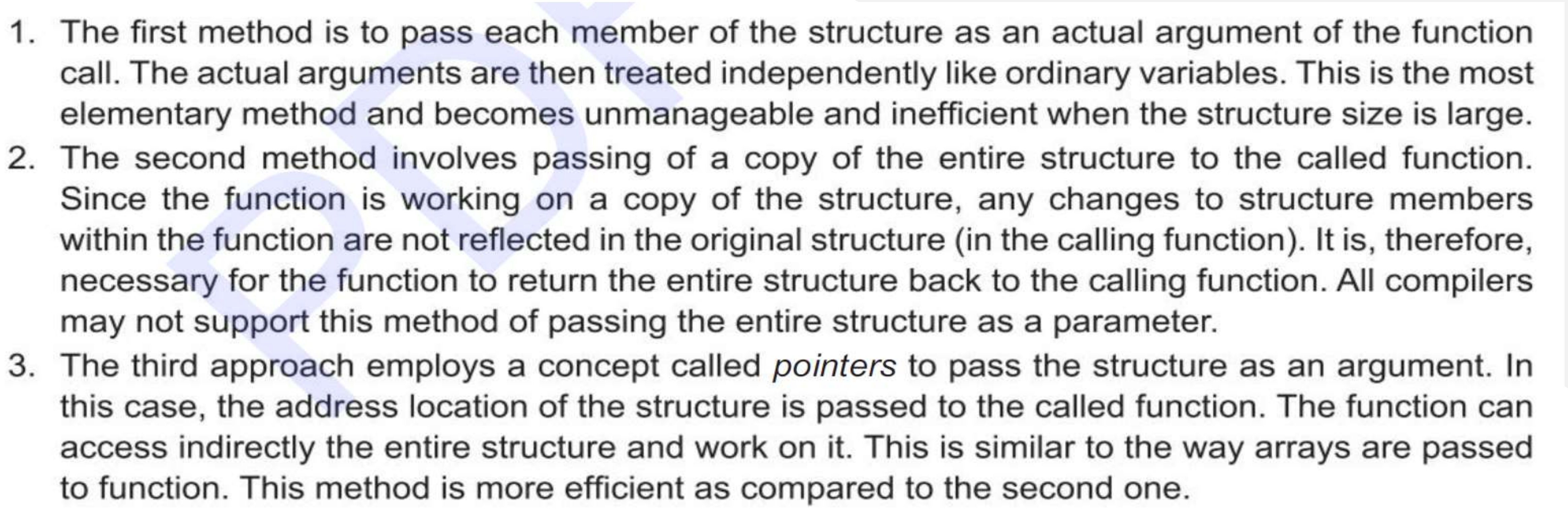
Structure and Function

- In C, we can work with structures in conjunction with functions. Functions can operate on structures by accepting them as arguments, returning them as results, or both.
- This allows us to encapsulate operations related to structures and manipulate them in a modular and organized manner.
- The following example demonstrating how to use structures and functions together in C:

```
struct Point {
    int x;
    int y;
};

void initializePoint(struct Point *p, int x, int y) {
    p->x = x;
    p->y = y;
}
```

Structure and Function

- In C, there are three methods by which the values of a structure can be transferred from one function to another:



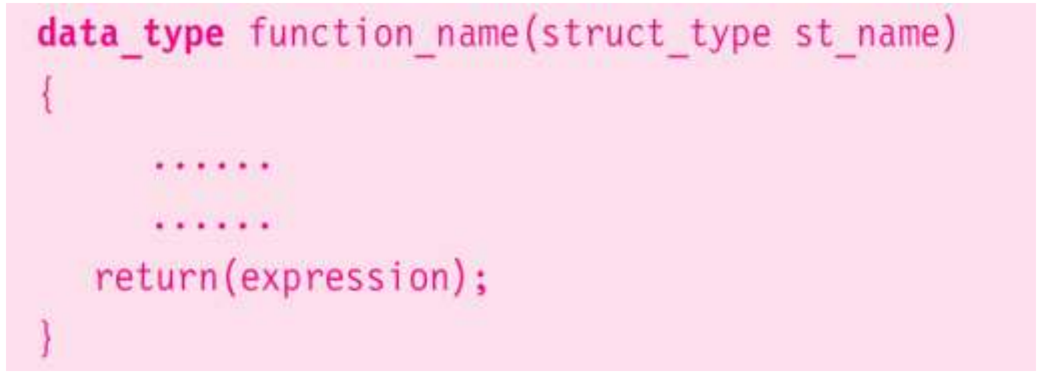
1. The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.
2. The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function. All compilers may not support this method of passing the entire structure as a parameter.
3. The third approach employs a concept called *pointers* to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way arrays are passed to function. This method is more efficient as compared to the second one.

Structure and Function

- The general format of sending a copy of a structure to the called function is:

  Function-name(struct-variable-name);

- The called function takes the following form:



```
data_type function_name(struct_type st_name)
{
        ......
        ......
    return(expression);
}
```

Structure and Function

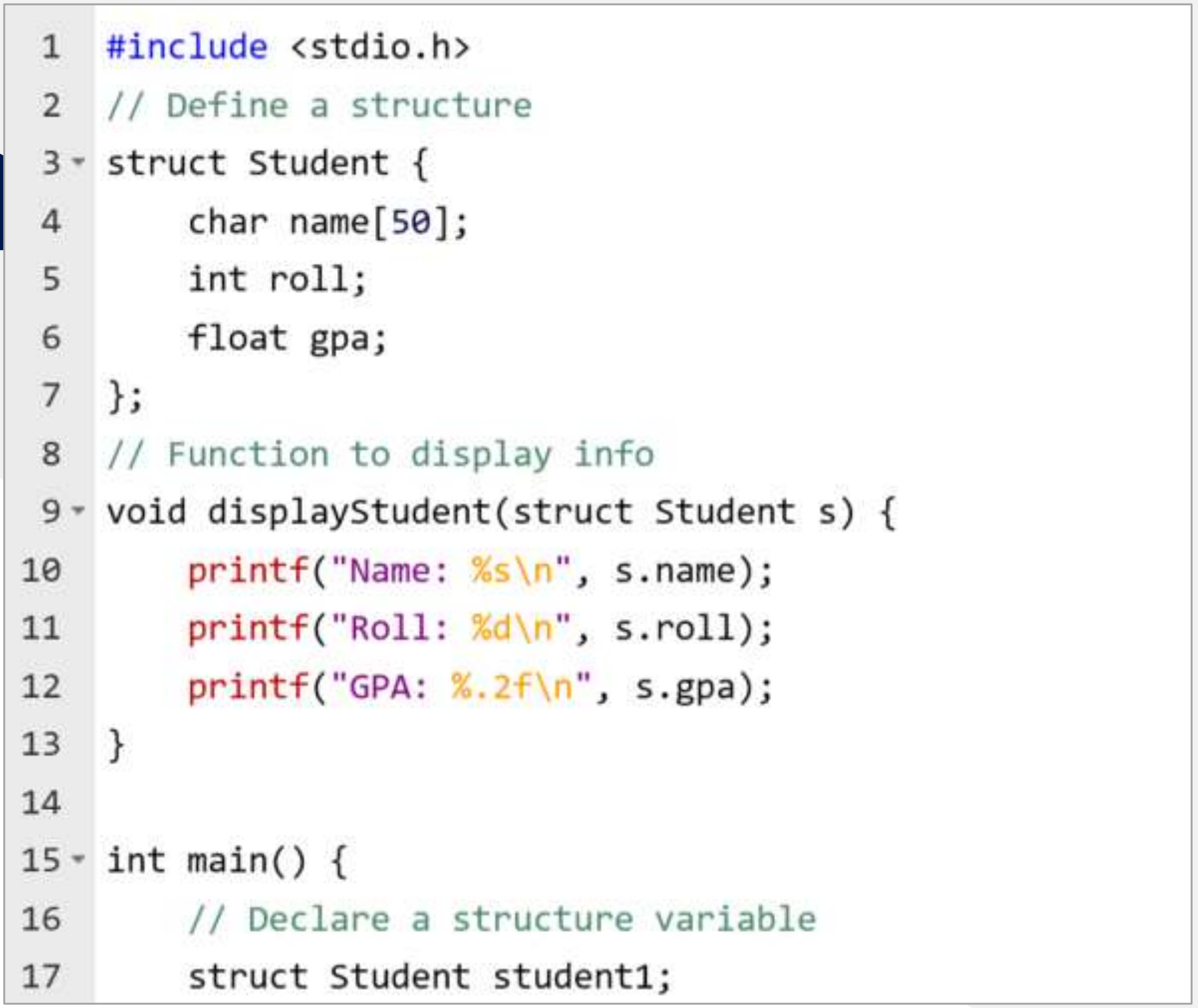
```
data_type function_name(struct_type st_name)
{
      ......
      ......
   return(expression);
}
```

The following points are important to note:

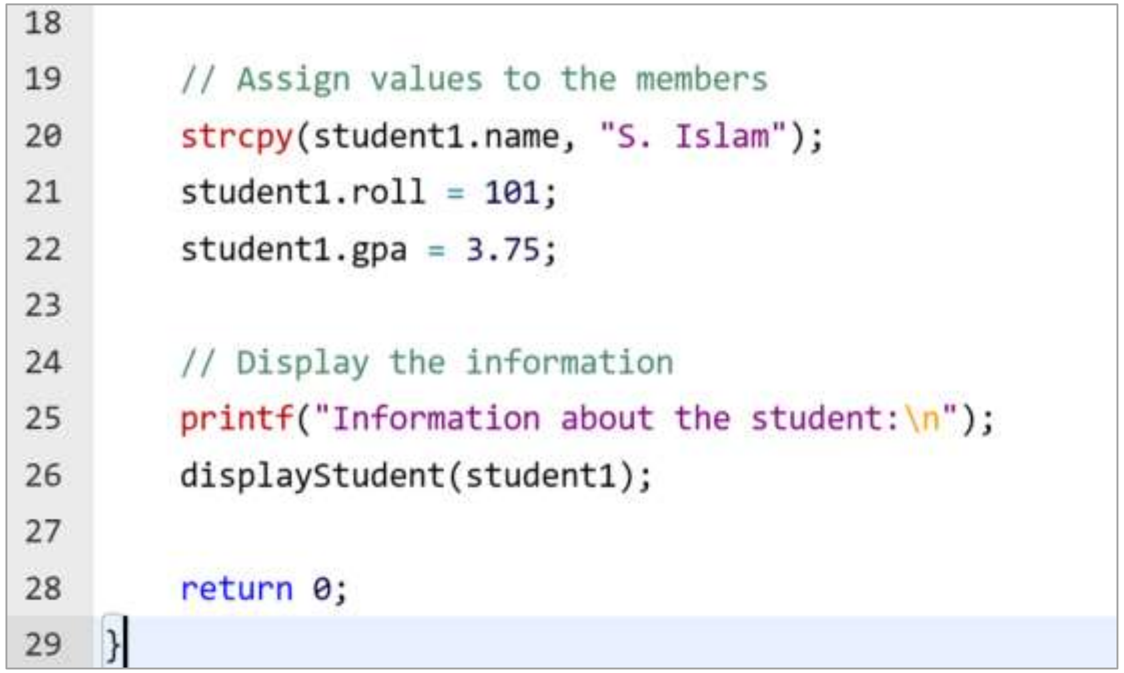
1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as **struct** with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same **struct** type.
3. The **return** statement is necessary only when the function is returning some data back to the calling function. The *expression* may be any simple variable or structure variable or an expression using simple variables.
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called functions must be declared in the calling function appropriately.

Structure and Function

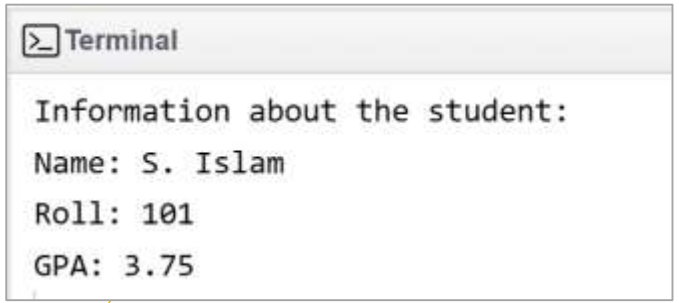
- Here's a simple C program that illustrates passing an entire structure as a parameter to a function:



```c
#include <stdio.h>
// Define a structure
struct Student {
    char name[50];
    int roll;
    float gpa;
};
// Function to display info
void displayStudent(struct Student s) {
    printf("Name: %s\n", s.name);
    printf("Roll: %d\n", s.roll);
    printf("GPA: %.2f\n", s.gpa);
}

int main() {
    // Declare a structure variable
    struct Student student1;
```



```c

    // Assign values to the members
    strcpy(student1.name, "S. Islam");
    student1.roll = 101;
    student1.gpa = 3.75;

    // Display the information
    printf("Information about the student:\n");
    displayStudent(student1);

    return 0;
}
```

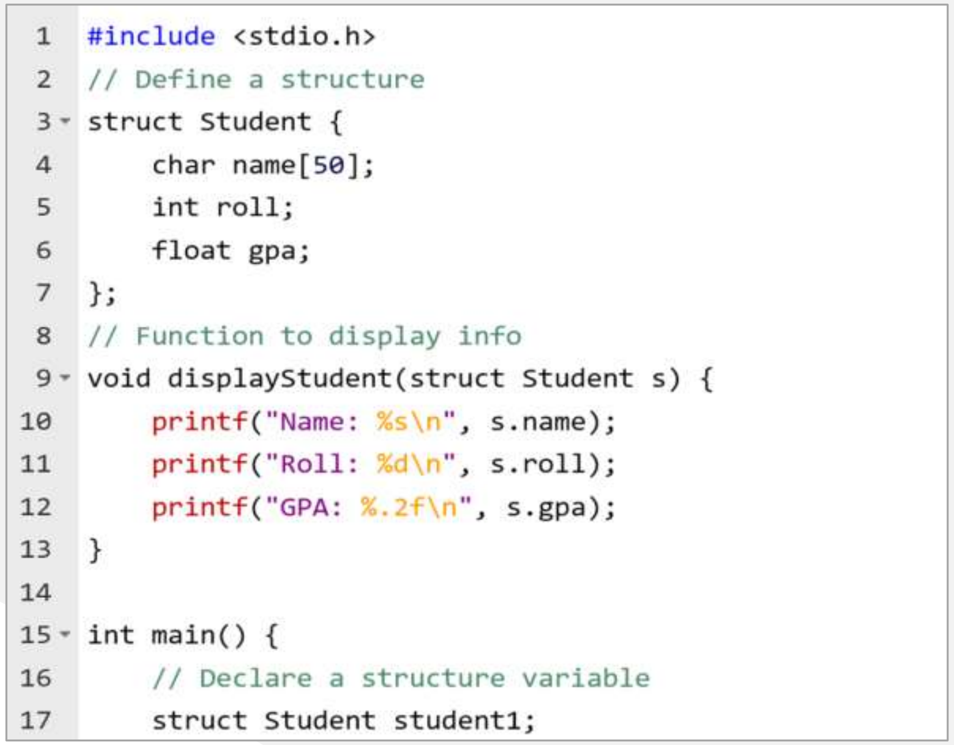


Terminal

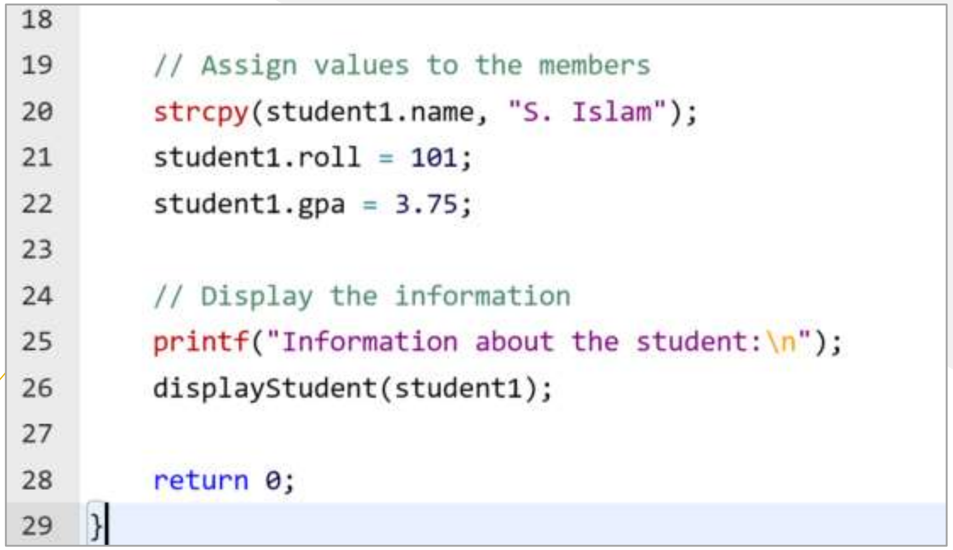
```
Information about the student:
Name: S. Islam
Roll: 101
GPA: 3.75
```

Structure and Function

- This demonstrates passing the entire structure as a parameter to a function in C. In this program:
  - We define a structure struct Student representing a student with three members: name, roll, and gpa.
  - We define a function displayStudent that takes a struct Student as a parameter and displays the information of the student contained in that structure.
  - In the main function, we declare a variable student1 of type struct Student.
  - We assign values to the members of student1.
  - We then call the displayStudent function and pass student1 as an argument.



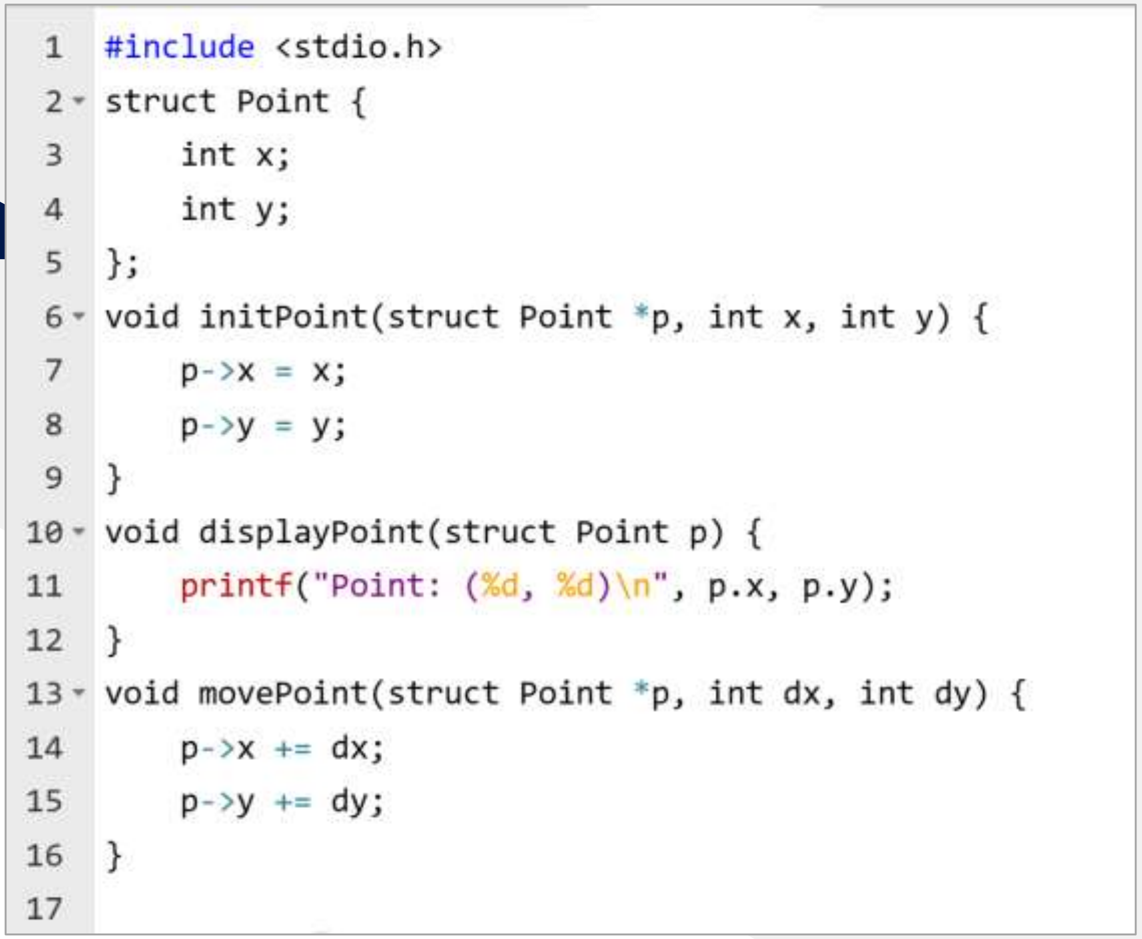
```c
#include <stdio.h>
// Define a structure
struct Student {
    char name[50];
    int roll;
    float gpa;
};
// Function to display info
void displayStudent(struct Student s) {
    printf("Name: %s\n", s.name);
    printf("Roll: %d\n", s.roll);
    printf("GPA: %.2f\n", s.gpa);
}

int main() {
    // Declare a structure variable
    struct Student student1;
```



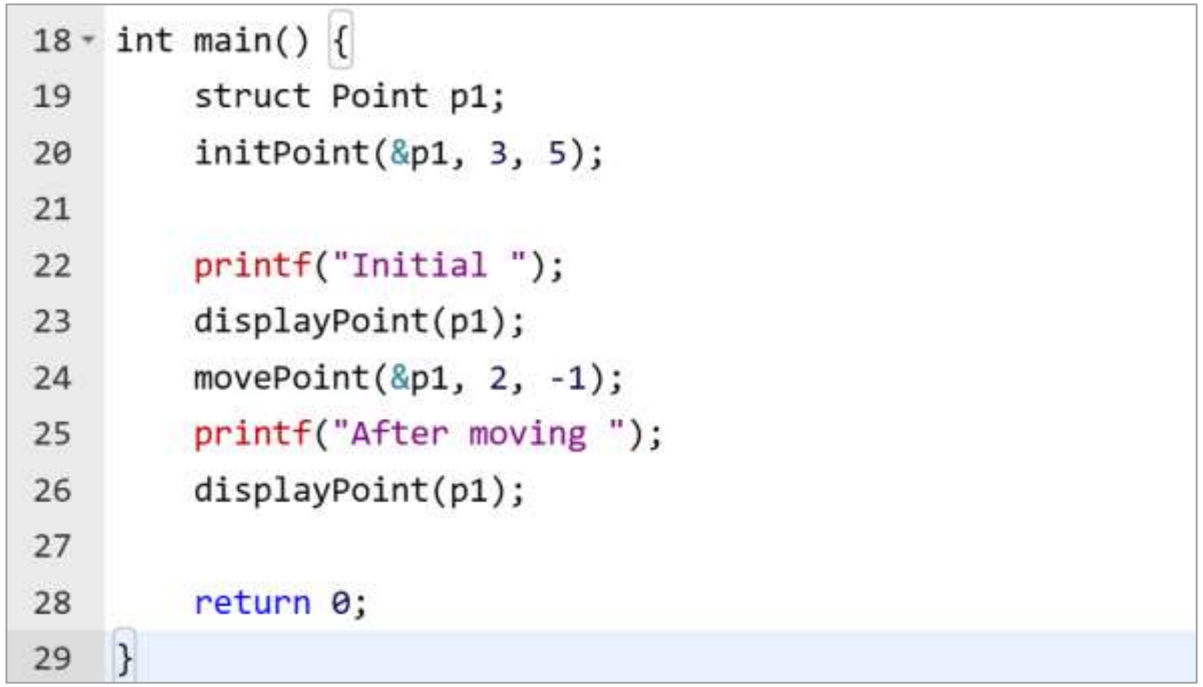
```c

    // Assign values to the members
    strcpy(student1.name, "S. Islam");
    student1.roll = 101;
    student1.gpa = 3.75;

    // Display the information
    printf("Information about the student:\n");
    displayStudent(student1);

    return 0;
}
```

Structure and Function

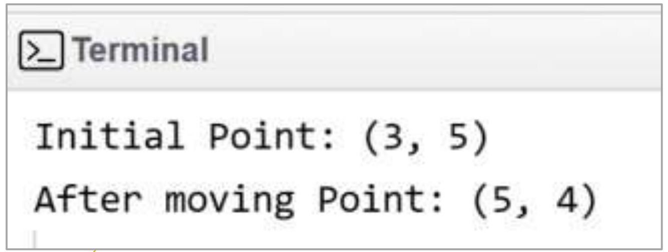
- Example of demonstrating how to use structures and functions together in C:



```c
#include <stdio.h>
struct Point {
    int x;
    int y;
};
void initPoint(struct Point *p, int x, int y) {
    p->x = x;
    p->y = y;
}
void displayPoint(struct Point p) {
    printf("Point: (%d, %d)\n", p.x, p.y);
}
void movePoint(struct Point *p, int dx, int dy) {
    p->x += dx;
    p->y += dy;
}

```



```c
int main() {
    struct Point p1;
    initPoint(&p1, 3, 5);

    printf("Initial ");
    displayPoint(p1);
    movePoint(&p1, 2, -1);
    printf("After moving ");
    displayPoint(p1);

    return 0;
}
```



Terminal

```
Initial Point: (3, 5)
After moving Point: (5, 4)
```

Unions in C

- In C programming, a union is a user-defined data type that allows us to store different data types in the same memory location.
- Unlike structures, where each member has its own separate memory space, all members of a union share the same memory space. This means that only one member of the union can be used at a time.
- Like structure, a union can be declared using the keyword ‘union’ as follows:

```
union item{
        int m;
        float x;
        char c;
}code;
```

- This declares a variable **code** of type **union item**. The union contains three members, each with a different data type. However, we can use **only one of them at a time**.

Unions in C

- The union contains three members, each with a different data type. However, we can use **only one of them at a time**.
- This is due to the fact that only one location is allocated for a union variable, irrespective of its size, as shown below:

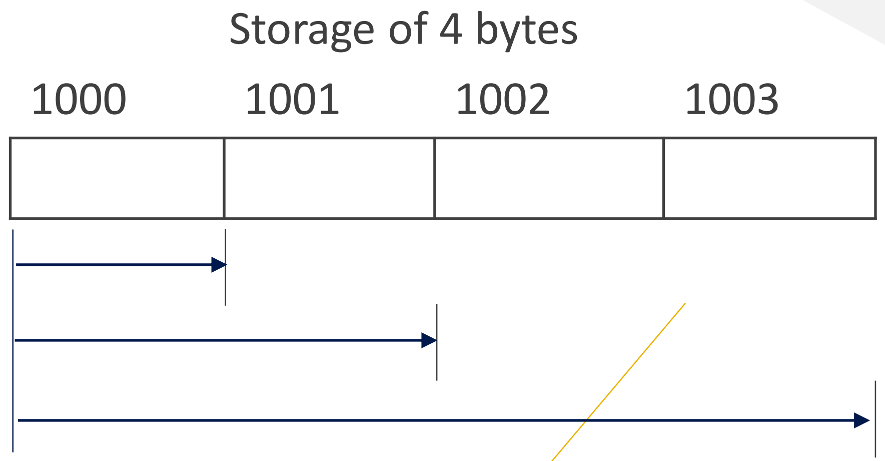


Fig: Sharing of a storage locating by union members.

Unions in C

- The above example shows how all the three variables share the same address. This assumes that a float variable requires 4 bytes of storage.
- To access a union member, we can use the same syntax that we use for structure members, as follows:

```
code.m
code.x
code.c
```

All are valid member variables.

Unions in C

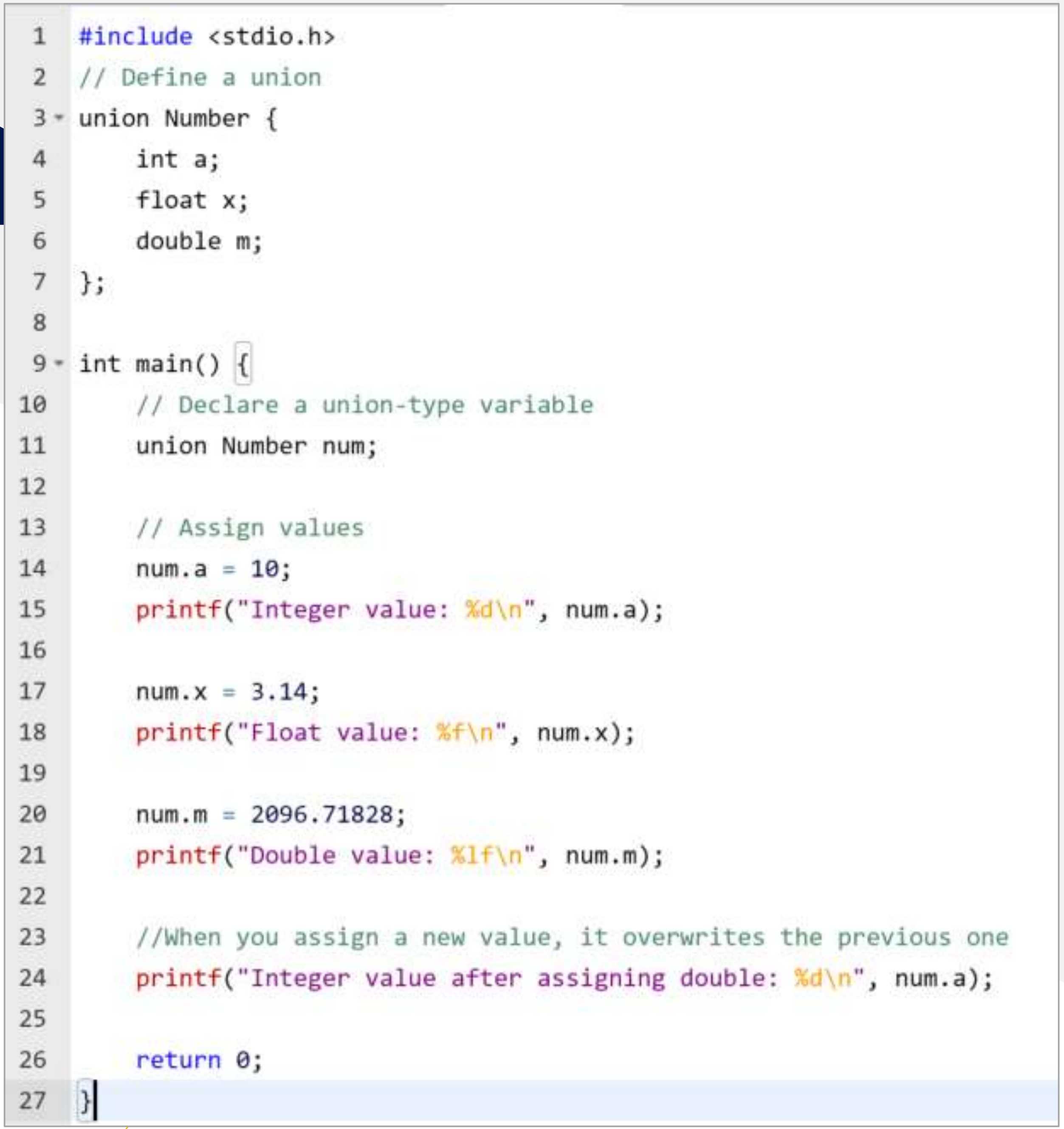
- During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the following statements:

```
code.m = 450;
code.x = 340.75;
printf(“m = %d \n x = %f \n”, code.m, code.x);
```

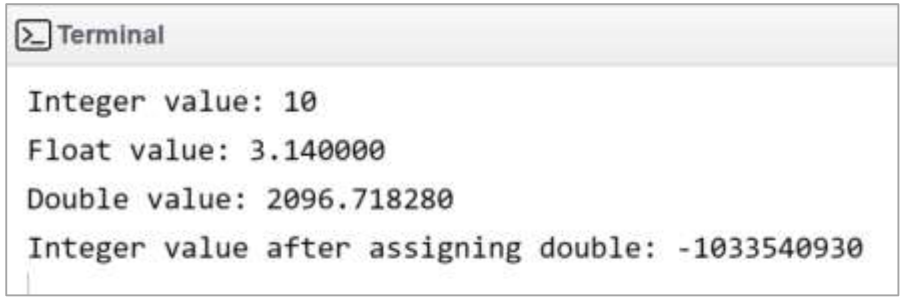
would produce the output.

Unions in C

- A basic example of how we can define and use a union in C:



```c
#include <stdio.h>
// Define a union
union Number {
    int a;
    float x;
    double m;
};

int main() {
    // Declare a union-type variable
    union Number num;

    // Assign values
    num.a = 10;
    printf("Integer value: %d\n", num.a);

    num.x = 3.14;
    printf("Float value: %f\n", num.x);

    num.m = 2096.71828;
    printf("Double value: %lf\n", num.m);

    //When you assign a new value, it overwrites the previous one
    printf("Integer value after assigning double: %d\n", num.a);

    return 0;
}
```



```
Integer value: 10
Float value: 3.140000
Double value: 2096.718280
Integer value after assigning double: -1033540930
```

?

THE END