ICT @ Mzuni

Introduction to programming with C

By: Chimango Nyasulu, PhD

Outline

- Arrays
- Multi-dimensional arrays
- Structures
- Unions

Arrays

- An array is a collection of items of the same data type stored in contiguous memory locations.

- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value. They provide a way to group related data together.

Key Features of Arrays

1. **Homogeneous Data Type**: All elements in an array must be of the same data type (e.g., all integers, all floats, etc.).

2. **Fixed Size:** The size of an array must be specified at the time of declaration and cannot be changed during runtime.

Arrays

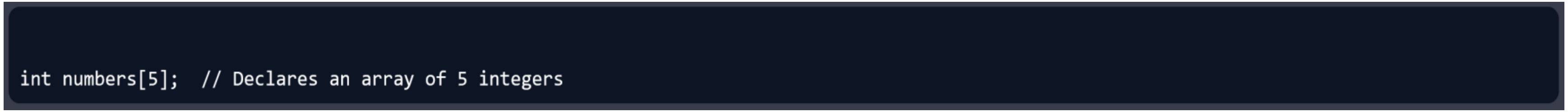
Key Features of Arrays

3. **Contiguous Memory Allocation**: Arrays are stored in consecutive memory locations, which allows for efficient access and manipulation of data.

4. **Indexing:** Elements in an array are accessed using indices, which start from 0. For example, in an array of size n, the valid indices are from 0 to n-1.

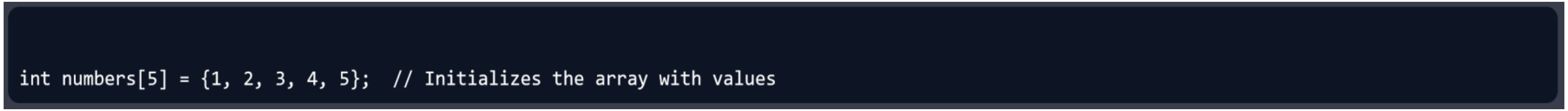
Declaration of Arrays:

- To declare an array in C, you specify the data type, the name of the array, and the size (number of elements) in square brackets.



```
int numbers[5];  // Declares an array of 5 integers
```

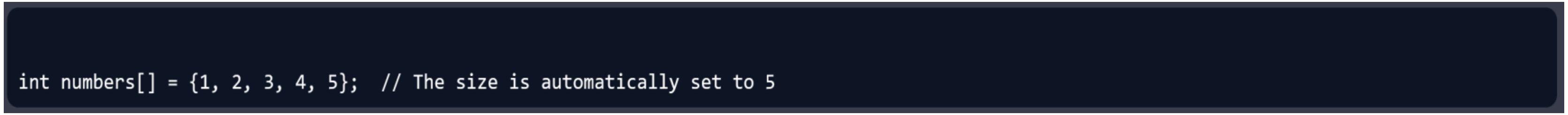
- You can initialize an array at the time of declaration.



```
int numbers[5] = {1, 2, 3, 4, 5};  // Initializes the array with values
```

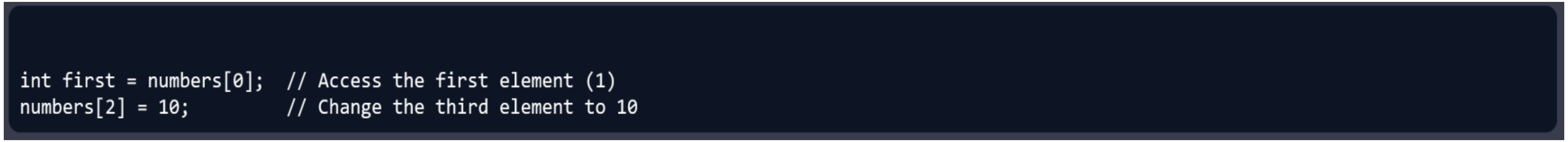
Arrays

- If the size is omitted, the compiler determines the size based on the number of initializers.



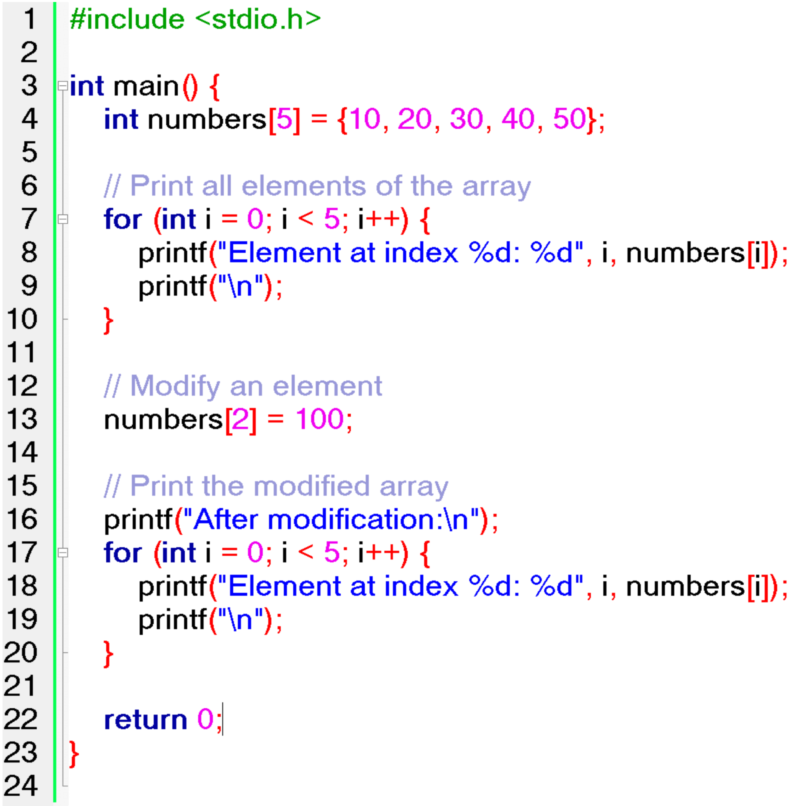
```
int numbers[] = {1, 2, 3, 4, 5};  // The size is automatically set to 5
```

- You can access or modify elements of an array using their index.



```
int first = numbers[0];  // Access the first element (1)
numbers[2] = 10;         // Change the third element to 10
```

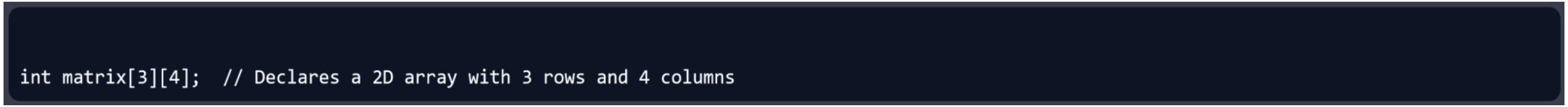
Arrays



```c
#include <stdio.h>

int main() {
    int numbers[5] = {10, 20, 30, 40, 50};

    // Print all elements of the array
    for (int i = 0; i < 5; i++) {
        printf("Element at index %d: %d", i, numbers[i]);
        printf("\n");
    }

    // Modify an element
    numbers[2] = 100;

    // Print the modified array
    printf("After modification:\n");
    for (int i = 0; i < 5; i++) {
        printf("Element at index %d: %d", i, numbers[i]);
        printf("\n");
    }

    return 0;
}

```

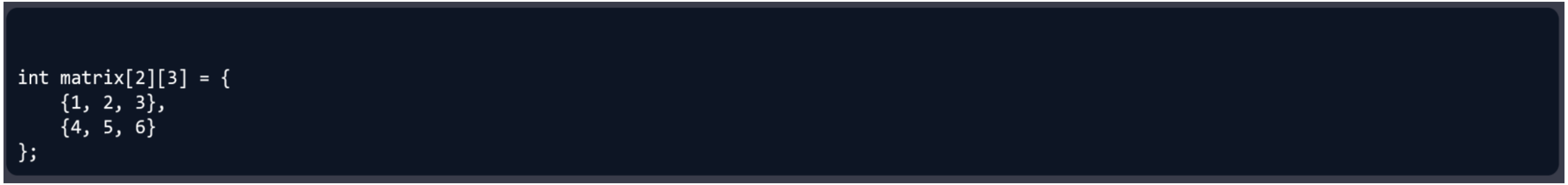
Multidimensional Arrays

- C also supports multidimensional arrays (like 2D arrays).



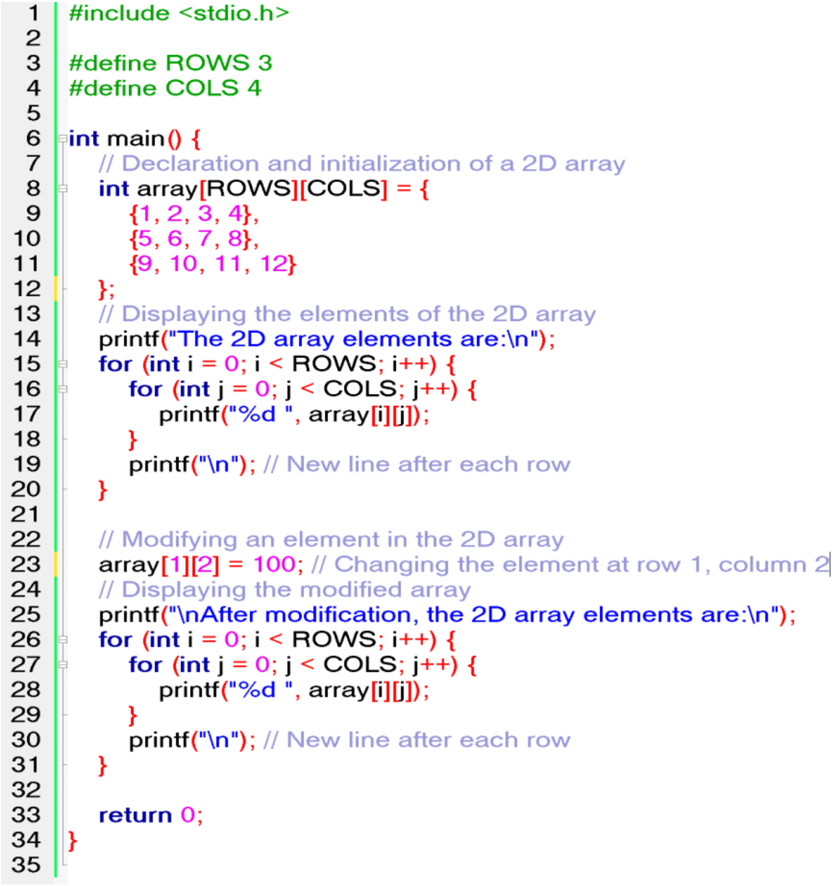
```c
int matrix[3][4];  // Declares a 2D array with 3 rows and 4 columns
```

- You can initialize a 2D array as follows:



```c
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Multidimensional Arrays



```c
#include <stdio.h>

#define ROWS 3
#define COLS 4

int main() {
    // Declaration and initialization of a 2D array
    int array[ROWS][COLS] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    // Displaying the elements of the 2D array
    printf("The 2D array elements are:\n");
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            printf("%d ", array[i][j]);
        }
        printf("\n"); // New line after each row
    }

    // Modifying an element in the 2D array
    array[1][2] = 100; // Changing the element at row 1, column 2
    // Displaying the modified array
    printf("\nAfter modification, the 2D array elements are:\n");
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            printf("%d ", array[i][j]);
        }
        printf("\n"); // New line after each row
    }

    return 0;
}
```

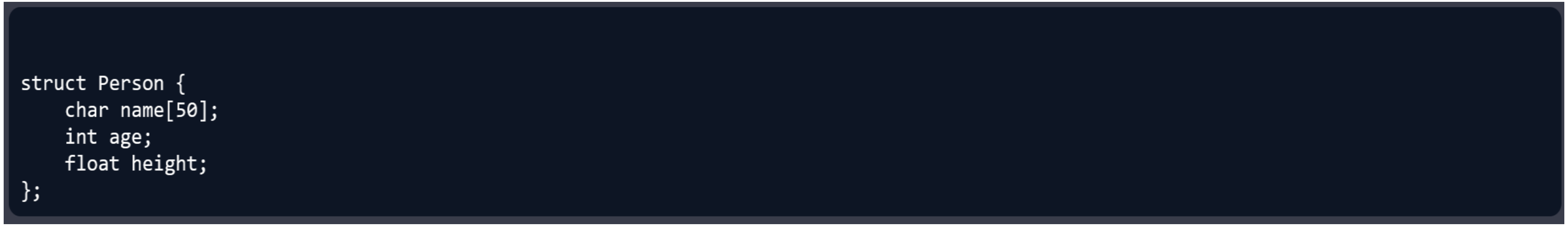
Structures

- A structure (often abbreviated as "struct") is a user-defined data type that allows the combination of data items of different kinds.

- Structures are used to group related variables (of different data types) together under a single name, which can be beneficial for organizing complex data.

Key features of Structures

1. **Definition:** A structure is defined using the struct keyword, followed by a name for the structure and a list of its members enclosed in curly braces.

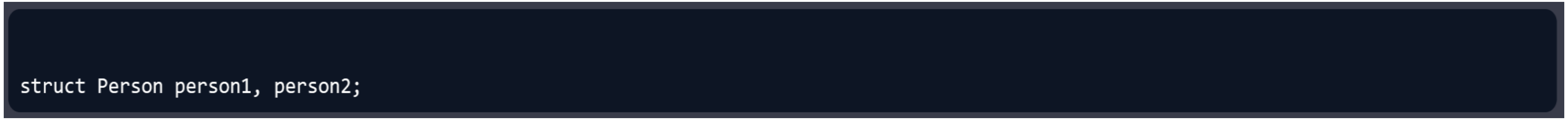


```
struct Person {
    char name[50];
    int age;
    float height;
};
```

Structures

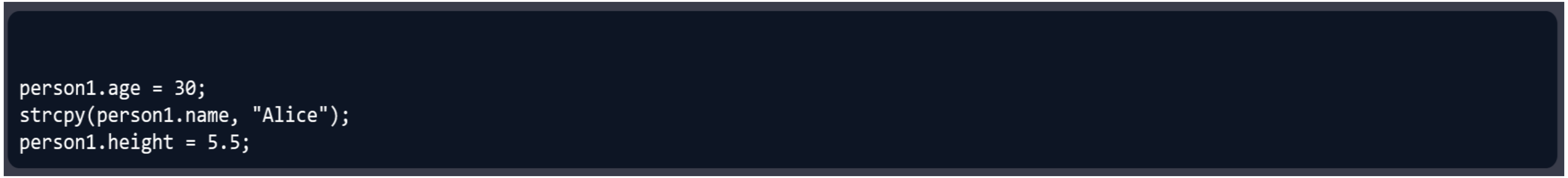
Key features of Structures

2. **Declaring Structure Variables:** After defining a structure, you can create variables of that structure type.



```c
struct Person person1, person2;
```

3. **Accessing Members:** You can access the members of a structure using the dot operator (.) for structure variables

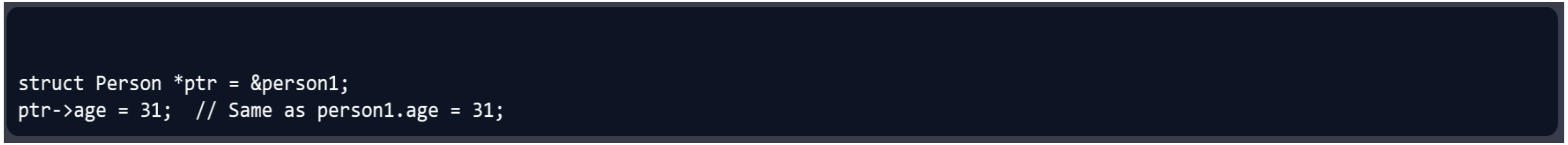


```c
person1.age = 30;
strcpy(person1.name, "Alice");
person1.height = 5.5;
```

Structures

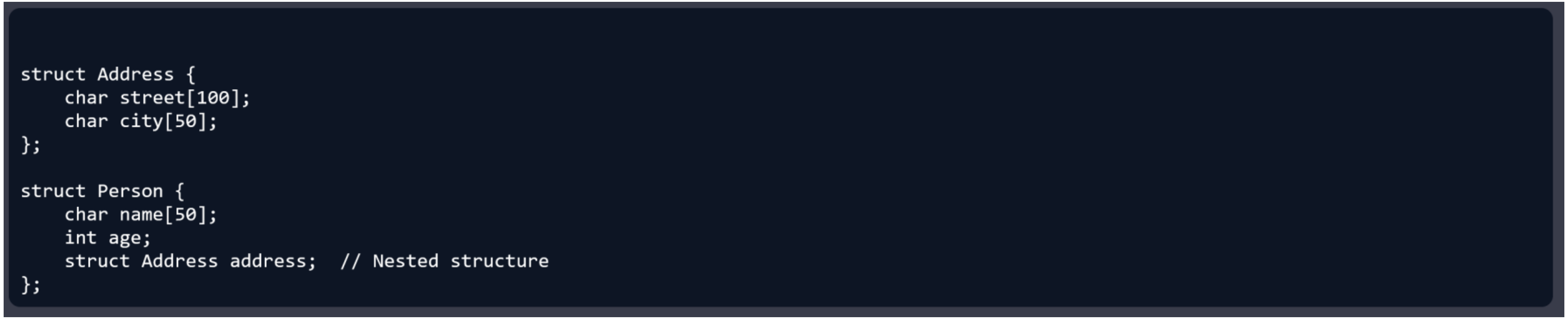
Key features of Structures

4. **Pointers to Structures:** You can also create pointers to structures and access members using the arrow operator (->).



```
struct Person *ptr = &person1;
ptr->age = 31;  // Same as person1.age = 31;
```

5. **Nested Structures:** Structures can contain other structures as members, allowing for more complex data representations.

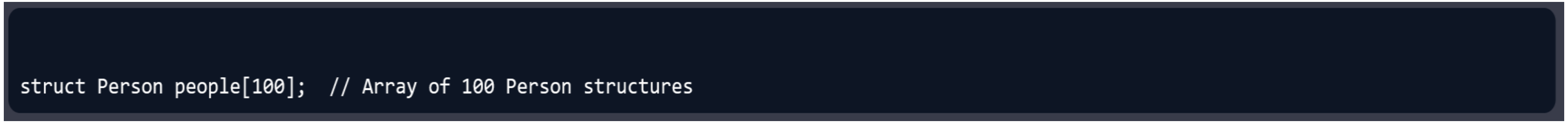


```
struct Address {
    char street[100];
    char city[50];
};

struct Person {
    char name[50];
    int age;
    struct Address address;  // Nested structure
};
```

Structures

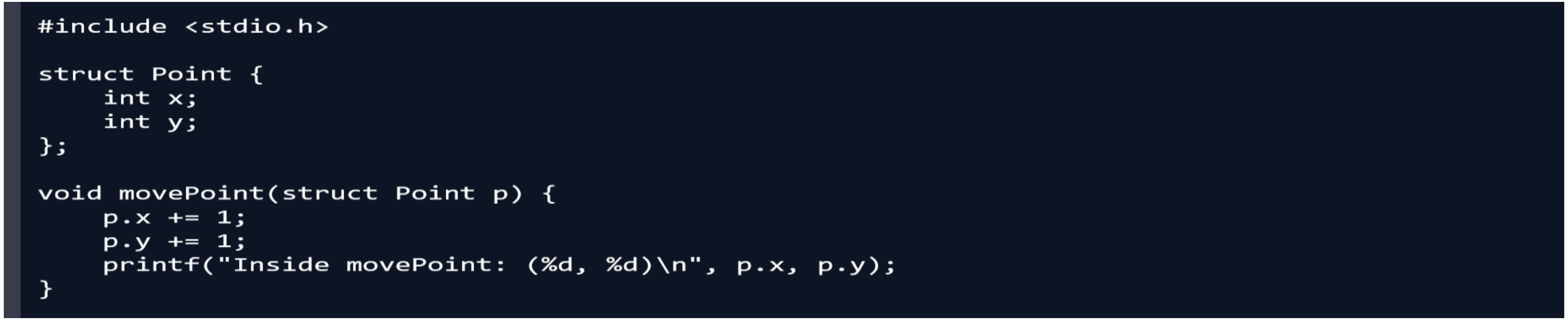
Key features of Structures

6. **Arrays of Structures:** You can create arrays of structures to store multiple records of the same type.



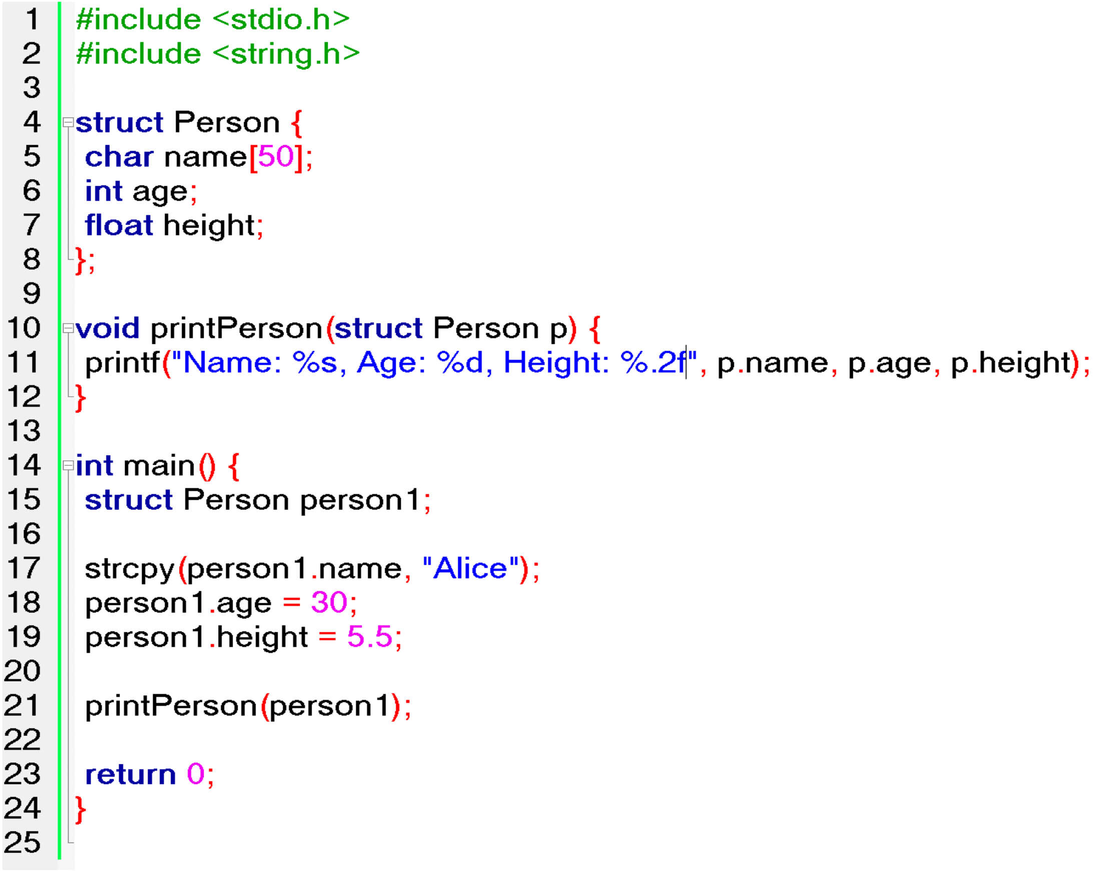
```
struct Person people[100];  // Array of 100 Person structures
```

7. **Passing Structures to Functions:** Structures can be passed to functions either by value or by reference (using pointers).



```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

void movePoint(struct Point p) {
    p.x += 1;
    p.y += 1;
    printf("Inside movePoint: (%d, %d)\n", p.x, p.y);
}
```

Structures



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

struct Person {
 char name[50];
 int age;
 float height;
};

void printPerson(struct Person p) {
 printf("Name: %s, Age: %d, Height: %.2f", p.name, p.age, p.height);
}

int main() {
 struct Person person1;

 strcpy(person1.name, "Alice");
 person1.age = 30;
 person1.height = 5.5;

 printPerson(person1);

 return 0;
}

```

Unions

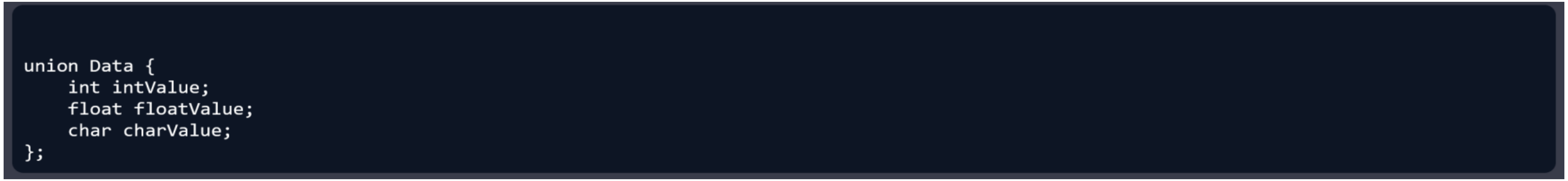
- A union is a special data structure that allows you to store different data types in the same memory location.

- A union can hold only one of its non-static data members at a time. This means that the size of the union is determined by the size of its largest member, and all members share the same memory space.

Key Features of Unions

1. **Memory Efficiency:** Since all members share the same memory, unions can be more memory-efficient than structures (structs), which allocate separate memory for each member.

2. **Declaration:** A union is declared using the union keyword, similar to a structure.



```
union Data {
    int intValue;
    float floatValue;
    char charValue;
};
```

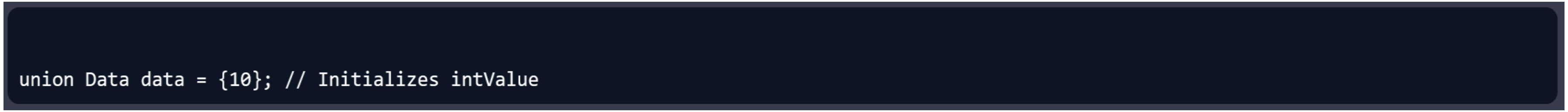
Unions

Key Features of Unions

3. **Usage:** You can create a variable of the union type and access its members. However, you should only use one member at a time, as writing to one member will overwrite the value of the others.

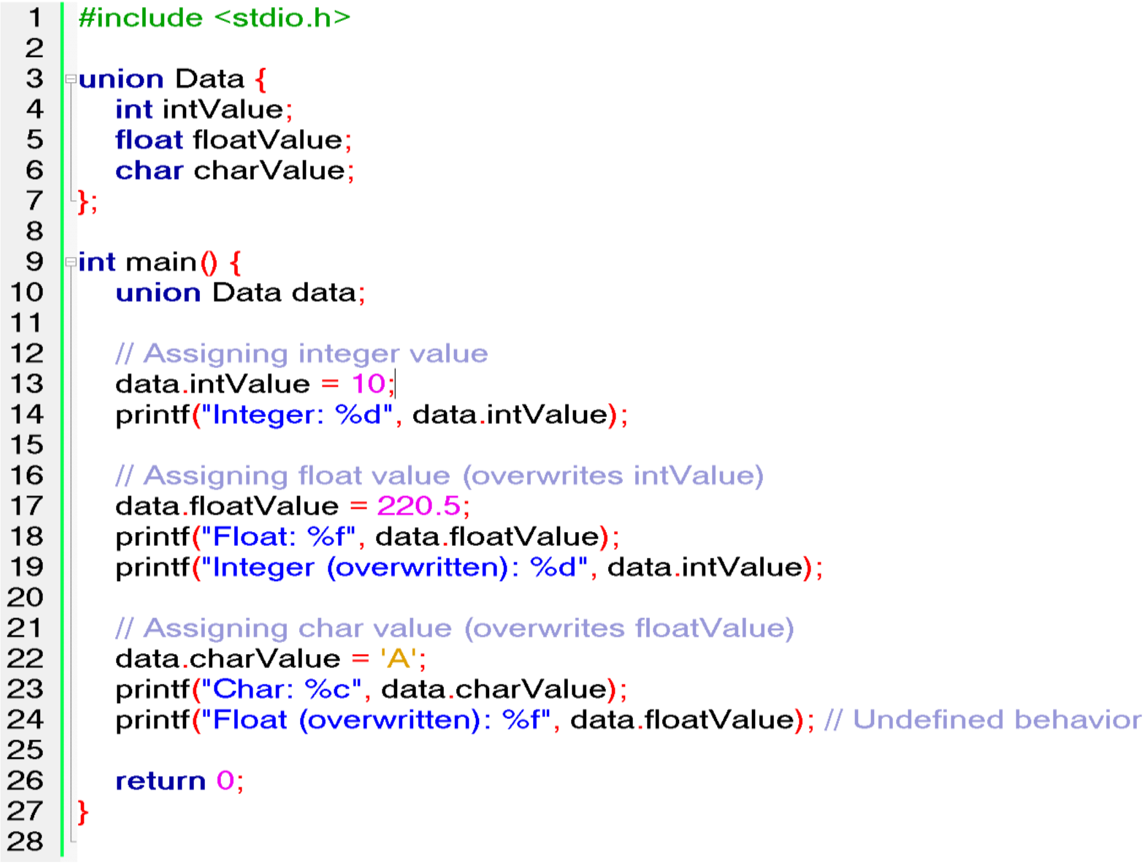
```
union Data data;
data.intValue = 10;
printf("%d", data.intValue); // Output: 10
```

4. **Initialization:** You can initialize a union at the time of declaration, but only the first member can be initialized directly.



```
union Data data = {10}; // Initializes intValue
```

Unions



```c
#include <stdio.h>

union Data {
    int intValue;
    float floatValue;
    char charValue;
};

int main() {
    union Data data;

    // Assigning integer value
    data.intValue = 10;
    printf("Integer: %d", data.intValue);

    // Assigning float value (overwrites intValue)
    data.floatValue = 220.5;
    printf("Float: %f", data.floatValue);
    printf("Integer (overwritten): %d", data.intValue);

    // Assigning char value (overwrites floatValue)
    data.charValue = 'A';
    printf("Char: %c", data.charValue);
    printf("Float (overwritten): %f", data.floatValue); // Undefined behavior

    return 0;
}

```