# One-Week-C-Primer (Day 5)

## Day 5: Strings , Structures and Unions

**Note : Refer to the Attached Document along with this for the codes**

### Strings

**Strings in C:**

In C, a string is an array of characters terminated by a null character (`'0'`). Strings are widely used for representing text or sequences of characters in C programming.

**Declaring and Initializing Strings:**

```
#include <stdio.h>

int main() {
    // Method 1: Declaring a string using an array of characters
    char str1[6] = {'H', 'e', 'l', 'l', 'o', '\\0'};

    // Method 2: Declaring a string using a string literal (null character is automaticall
y added)
    char str2[] = "Hello";

    printf("str1: %s\n", str1); // Output: Hello
    printf("str2: %s\n", str2); // Output: Hello

    return 0;
}
```

**String Input and Output:**

```
#include <stdio.h>

int main() {
    char str[50];

    // Input a string from the user
    printf("Enter a string: ");
    scanf("%s", str);

    // Output the entered string
```

```
    printf("You entered: %s\n", str);

    return 0;
}
```

**String Functions:**

C provides several string manipulation functions in the standard library (`<string.h>`) for working with strings.

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

int main() {
    char str1[20] = "Hello";
    char str2[] = " World";

    // Concatenate str2 to str1
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1); // Output: Hello World

    // Get the length of the string
    int length = strlen(str1);
    printf("Length of the string: %d\n", length); // Output: 11

    // Compare two strings
    int result = strcmp(str1, "Hello World");
    if (result == 0) {
        printf("The strings are equal.\n");
    } else {
        printf("The strings are not equal.\n");
    }

    return 0;
}
```

**Important Points about Strings in C:**

1. In C, a string is an array of characters. Therefore, you can use array indexing to access individual characters in the string.
2. String literals (e.g., "Hello") are stored in read-only memory, and modifying them directly can lead to undefined behavior.
3. Strings must be null-terminated (`'\0'` at the end) to be treated as valid C strings. Null termination allows C functions to determine the end of the string.

4. Always use proper format specifiers (`%s`) when reading and printing strings to prevent buffer overflow issues.
5. C does not have a built-in string type, so strings are represented as arrays of characters. This lack of a string type means that you need to be careful when working with strings to avoid buffer overflows and other memory-related issues.
6. When working with strings, be mindful of the length of the strings and ensure that there is enough space in the destination buffer to hold the concatenated or modified string.
7. The standard library `<string.h>` provides many useful functions for string manipulation, such as `strcpy()`, `strncpy()`, `strcat()`, `strncat()`, `strlen()`, `strcmp()`, `strncmp()`, `strstr()`, etc.

Understanding strings and using appropriate string functions in C is vital for effectively working with textual data in programs. Always ensure proper memory management and boundary checks when working with strings to avoid common bugs and vulnerabilities.

---

## Structures in C

Structures in C are user-defined data types that allow you to group multiple variables of different types under a single name. A structure is a composite data type that can represent a real-world entity with various attributes.

**Declaring a Structure:**

```
#include <stdio.h>

// Structure declaration
struct Student {
    char name[50];
    int age;
    float marks;
};

int main() {
    // Structure variable declaration
    struct Student student1;

    // Accessing structure members using the dot operator
    strcpy(student1.name, "John");
```

```
    student1.age = 20;
    student1.marks = 85.5;

    // Printing structure members
    printf("Name: %s\n", student1.name);
    printf("Age: %d\n", student1.age);
    printf("Marks: %.2f\n", student1.marks);

    return 0;
}
```

**Nested Structures:**

Structures can be nested inside other structures, allowing you to create complex data structures.

```
#include <stdio.h>

struct Address {
    char city[30];
    char state[30];
};

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

int main() {
    struct Employee emp1;

    strcpy(emp1.name, "Alice");
    emp1.age = 30;
    strcpy(emp1.address.city, "New York");
    strcpy(emp1.address.state, "NY");

    printf("Name: %s\n", emp1.name);
    printf("Age: %d\n", emp1.age);
    printf("City: %s\n", emp1.address.city);
    printf("State: %s\n", emp1.address.state);

    return 0;
}
```

**Typedef with Structures:**

You can use `typedef` to create a new name for a structure, making it easier to declare structure variables.

```
#include <stdio.h>

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

int main() {
    Point p1 = {10, 20};

    printf("Coordinates: (%d, %d)n", p1.x, p1.y);

    return 0;
}
```

**Passing Structures to Functions:**

You can pass structures to functions as arguments, either by value or by reference (using pointers).

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

void printPoint(struct Point p) {
    printf("Coordinates: (%d, %d)\n", p.x, p.y);
}

int main() {
    struct Point p1 = {5, 8};
    printPoint(p1);

    return 0;
}
```

**Returning Structures from Functions:**

Functions can also return structures.

```
#include <stdio.h>

struct Rectangle {
    int length;
```

```
    int width;
};

struct Rectangle createRectangle(int l, int w) {
    struct Rectangle r;
    r.length = l;
    r.width = w;
    return r;
}

int main() {
    struct Rectangle rect = createRectangle(10, 5);
    printf("Rectangle: length=%d, width=%d\n", rect.length, rect.width);

    return 0;
}
```

**Arrays of Structures:**

You can create arrays of structures to represent a collection of entities.

```
#include <stdio.h>

struct Book {
    char title[100];
    char author[50];
    int pages;
};

int main() {
    struct Book library[3];

    // Input book details
    for (int i = 0; i < 3; i++) {
        printf("Enter title: ");
        scanf("%s", library[i].title);
        printf("Enter author: ");
        scanf("%s", library[i].author);
        printf("Enter number of pages: ");
        scanf("%d", &library[i].pages);
    }

    // Display book details
    printf("\nLibrary Contents:\n");
    for (int i = 0; i < 3; i++) {
        printf("Book %d:\n", i + 1);
        printf("Title: %s\n", library[i].title);
        printf("Author: %s\n", library[i].author);
        printf("Pages: %d\n", library[i].pages);
        printf("\\n");
    }
```

```
    return 0;
}
```

Structures in C allow you to organize and manage related data efficiently. They are essential for creating complex data structures and handling real-world entities in programs. By using structures, you can achieve better code organization, readability, and reusability.

---

## Union in C

Unions in C are user-defined data types that allow you to store different data types in the same memory location. Unlike structures, where each member has its own memory location, all members of a union share the same memory space. Unions are useful when you need to represent different data types using the same memory block and access only one member at a time.

**Declaring a Union:**

```
#include <stdio.h>

// Union declaration
union Data {
    int intValue;
    float floatValue;
    char stringValue[20];
};

int main() {
    union Data data;

    data.intValue = 42;
    printf("Int Value: %d\n", data.intValue);

    data.floatValue = 3.14;
    printf("Float Value: %.2f\n", data.floatValue);

    strcpy(data.stringValue, "Hello");
    printf("String Value: %s\n", data.stringValue);

    return 0;
}
```

**Accessing Union Members:**

Since all members share the same memory, changing one member affects the other members. You should only access the appropriate member based on the type stored in the union.

```
#include <stdio.h>

union Data {
    int intValue;
    float floatValue;
    char stringValue[20];
};

int main() {
    union Data data;
    data.intValue = 42;

    printf("Int Value: %d\n", data.intValue);
    printf("Float Value: %.2f\n", data.floatValue); // Undefined behavior
    printf("String Value: %s\n", data.stringValue); // Undefined behavior

    return 0;
}
```

**Size of Unions:**

The size of a union is determined by the size of its largest member, as all members share the same memory. Unions are space-efficient when you need to store data of different types but use only one at a time.

```
#include <stdio.h>

union Data {
    int intValue;
    float floatValue;
    char stringValue[20];
};

int main() {
    union Data data;
    printf("Size of Union: %zu bytes\n", sizeof(data)); // Output depends on the largest m
ember's size

    return 0;
}
```

**Unions vs. Structures:**

- In structures, each member has its own memory space, while in unions, all members share the same memory.
- A union is useful when you need to store different data types and save memory. A structure is used to group related data together.
- In unions, only one member should be accessed at a time, as the value of other members might be undefined or corrupted.
- Structures are suitable for representing entities with multiple attributes, whereas unions are suitable for representing data with alternative interpretations.

**Unions with Pointers:**

You can use pointers to access union members dynamically.

```
#include <stdio.h>

union Data {
    int intValue;
    float floatValue;
    char stringValue[20];
};

int main() {
    union Data data;
    union Data *ptr = &data;

    ptr->intValue = 42;
    printf("Int Value: %d\n", ptr->intValue);

    ptr->floatValue = 3.14;
    printf("Float Value: %.2f\n", ptr->floatValue);

    strcpy(ptr->stringValue, "Hello");
    printf("String Value: %s\n", ptr->stringValue);

    return 0;
}
```

Unions are a powerful feature in C when used correctly, but they require careful handling due to their shared memory nature. They are useful in situations where you need to conserve memory by representing different data types in the same memory block. However, be cautious when

accessing union members and ensure that you use the appropriate member based on the type stored in the union.

---