UNIT III: Arrays and Strings

`Arrays indexing, memory model, programs with array of integers, two dimensional arrays, Introduction to Strings.`

What is an Array?

An **array** is a collection of items stored at contiguous memory locations. It is a data structure that can hold a fixed number of elements of the same data type. Arrays are commonly used in programming to store multiple values in a single variable, which can be accessed using an index.

Key Characteristics of Arrays:

- Fixed size: The size of an array must be defined at the time of declaration.
- Homogeneous: All elements in an array must be of the same data type.
- Random access: Elements can be accessed using their index.

Types of Arrays

1. **One-Dimensional Array**: A linear array that stores a collection of elements in a single row or column.

   Example in C:

   ```
   int numbers[5] = {10, 20, 30, 40, 50};
   ```

2. **Two-Dimensional Array**: An array of arrays, which can be visualized as a matrix with rows and columns.

   Example in C:

   ```
   int matrix[3][3] = {
       {1, 2, 3},
       {4, 5, 6},
       {7, 8, 9}
   };
   ```

3. **Multi-Dimensional Array**: An extension of two-dimensional arrays, which can have more than two dimensions.

   Example in C:

   ```
   int tensor[2][3][4]; // A three-dimensional array
   ```

4. **Dynamic Array**: An array whose size can be determined during runtime, typically implemented using pointers.

   Example in C:

   ```
   int *dynamicArray;
   int size = 5;
   dynamicArray = (int *)malloc(size * sizeof(int));
   ```

What is a String?

A **string** is a sequence of characters terminated by a null character (`'\0'`). In C, strings are represented as arrays of characters.

Representing a String with Array and Pointer

1. **Using an Array**:

   ```
   char str[6] = "Hello"; // The array size must be one more than the number
   of characters for the null terminator
   ```

2. **Using a Pointer**:

   ```
   char *str = "Hello"; // Pointer to a string literal
   ```

String Functions with Examples

Here are some common string functions in C from the `<string.h>` library:

1. **strlen**: Returns the length of a string.

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

   int main() {
       char str[] = "Hello";
       printf("Length of the string: %lu\n", strlen(str));
       return 0;
   }
   ```

2. **strcpy**: Copies one string to another.

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

   int main() {
       char source[] = "Hello";
       char destination[6]; // Make sure to allocate enough space
       strcpy(destination, source);
       printf("Copied string: %s\n", destination);
       return 0;
   }
   ```

3. **strcat**: Concatenates two strings.

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

   int main() {
       char str1[20] = "Hello, ";
       char str2[] = "World!";
       strcat(str1, str2);
       printf("Concatenated string: %s\n", str1);
       return 0;
   }
   ```

4. **strcmp**: Compares two strings.

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

```
int main() {
    char str1[] = "Hello";
    char str2[] = "World";
    int result = strcmp(str1, str2);
    if (result < 0) {
        printf("'%s' is less than '%s'\n", str1, str2);
    } else if (result > 0) {
        printf("'%s' is greater than '%s'\n", str1, str2);
    } else {
        printf("Both strings are equal.\n");
    }
    return 0;
}
```

These examples demonstrate how to work with arrays and strings in C, showcasing their characteristics, types, and common operations.

The **memory model** in programming refers to how a programming language or runtime environment allocates, manages, and organizes memory for storing data and code. Understanding the memory model is essential for efficient programming, particularly in languages like C and C++ where the programmer has direct control over memory management. Here's an overview of the memory model, including its various segments and concepts:

Memory Segments

1. **Stack**:

    - The stack is used for static memory allocation, where data is stored in a last-in, first-out (LIFO) order.
    - It is typically used for function call management, local variables, and control flow (return addresses).
    - Memory is allocated and deallocated automatically when functions are called and returned.
    - Size is generally limited, and excessive use (e.g., deep recursion) can lead to stack overflow.

    Example:

    ```
    void function() {
        int localVariable; // Allocated on the stack
    }
    ```

2. **Heap**:

    - The heap is used for dynamic memory allocation, where memory can be allocated and freed at any time during the program's execution.
    - It allows for flexible memory management, enabling data structures like linked lists, trees, and more.
    - The programmer must explicitly allocate (`malloc`, `calloc`) and deallocate (`free`) memory, leading to potential memory leaks if not managed properly.

    Example:

    ```
    int *dynamicArray = (int *)malloc(10 * sizeof(int)); // Allocated on the
    heap
    ```

```
free(dynamicArray); // Deallocated
```

3. **Data Segment**:

   - This segment is used for storing global and static variables.
   - It is further divided into:
     - **Initialized data segment**: Stores global and static variables that are initialized by the programmer.
     - **Uninitialized data segment (BSS)**: Stores global and static variables that are declared but not initialized.

   Example:

   ```
   int globalVariable = 5; // Initialized data segment
   static int staticVariable; // BSS segment (uninitialized)
   ```

4. **Text Segment**:

   - The text segment contains the compiled code (instructions) of the program.
   - It is typically read-only to prevent accidental modification of the code during execution.

   Example:

   ```
   void exampleFunction() {
       // Code executed by the CPU
   }
   ```

Memory Layout Overview

A typical memory layout for a C/C++ program might look like this:

```
+---------------------+
|     Stack           |  (grows downward)
|                     |
|     Heap            |  (grows upward)
|                     |
|  Uninitialized Data |
| (BSS Segment)       |
|                     |
|  Initialized Data   |
| (Data Segment)      |
|                     |
|     Text Segment    |
|                     |
+---------------------+
```

Important Concepts

- **Memory Address**: Each byte of memory has a unique address that can be referenced.

- **Pointer**: A variable that stores the address of another variable. Pointers are essential for dynamic memory allocation and array manipulation.

  Example:

  ```
  int x = 10;
  int *p = &x; // p stores the address of x
  ```

- **Memory Management**: The process of allocating and freeing memory. It can be manual (using `malloc` and `free`) or automatic (using garbage collection in languages like Java).
- **Fragmentation**: Over time, dynamic memory allocation can lead to fragmentation, where free memory is divided into small, non-contiguous blocks, making it hard to allocate larger blocks.

<u>More Examples;</u>

```
#include <stdio.h>
#include<string.h>
#include <dos.h>
int main()
{
    int frequencies[] = {
        261,   // Sa (C)
        294,   // Re (D)
        329,   // Ga (E)
        349,   // Ma (F)
        392,   // Pa (G)
        440,   // Dha (A)
        493,   // Ni (B)
        523    // Sa (C, octave)
    },i;

   for (i=0;i<strlen(frequencies);i++)
   {
      sound(frequencies[i]);
      delay(100);
   }

   nosound();

   return 0;
}
```

UNIT IV: Pointers & User Defined Data types

Pointers, dereferencing and address operators, pointer and address arithmetic, array manipulation using pointers, User-defined data types-Structures and Unions.

1. Pointers

A **pointer** is a variable that stores the memory address of another variable. Pointers provide flexibility and efficiency in programming by allowing for dynamic memory management and manipulation of data stored in arrays and structures.

Declaration of Pointers:

```
int *p; // Declares a pointer to an integer
char *c; // Declares a pointer to a character
```

Example:

```
int x = 10;
int *p = &x; // p now stores the address of x
```

2. Dereferencing and Address Operators

- **Address Operator (&)**: Used to get the address of a variable.

  ```
  int x = 10;
  int *p = &x; // &x gives the address of x
  ```

- **Dereferencing Operator (*)**: Used to access or modify the value stored at the address a pointer is pointing to.

  ```
  int x = 10;
  int *p = &x;
  printf("%d", *p); // Output: 10
  ```

<u>Pointer to an Integer</u>

A pointer to an integer holds the address of an integer variable.

Example: Pointer to an Integer

```
#include <stdio.h>

int main() {
    int num = 42;
    int *ptr = &num; // Pointer to the integer variable

    printf("Integer: %d\n", *ptr); // Dereferencing the pointer to get the
integer

    return 0;
}
```

Output:

```
Integer: 42
```

Pointer to a Float:-

A pointer to a float holds the address of a float variable.

Example: Pointer to a Float

```
#include <stdio.h>

int main() {
    float f = 3.14f;
    float *ptr = &f; // Pointer to the float variable

    printf("Float: %.2f\n", *ptr); // Dereferencing the pointer to get the float

    return 0;
}
```

Output:

```
Float: 3.14
```

Pointer to a Character:-

A pointer to a character holds the address of a single character variable.

Example: Pointer to a Character

```
#include <stdio.h>

int main() {
    char ch = 'A';
    char *ptr = &ch; // Pointer to the character variable

    printf("Character: %c\n", *ptr); // Dereferencing the pointer to get the
character

    return 0;
}
```

Output:

```
Character: A
```

Pointer to a String

In C, a string is represented as an array of characters ending with a null character (`'\0'`). A pointer to a string points to the first character of that character array.

Example: Pointer to a String

```
#include <stdio.h>

int main() {
    char str[] = "Hello, World!";
    char *ptr = str; // Pointer to the first character of the string

    // Printing characters using the pointer
    while (*ptr != '\0') { // Loop until the null character
        printf("%c ", *ptr);
        ptr++; // Move to the next character
    }
    printf("\n");

    return 0;
}
```

Output:

```
H e l l o ,   W o r l d !
```

Pointer to an Array:-

A pointer to an array points to the first element of the array. You can use this pointer to access and manipulate the elements of the array.

Example: Pointer to an Array

```
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr; // Pointer to the first element of the array

    // Accessing array elements using the pointer
    for (int i = 0; i < 5; i++) {
        printf("Element %d: %d\n", i, *(ptr + i)); // Using pointer arithmetic
    }

    return 0;
}
```

Output:

```
Element 0: 10
Element 1: 20
Element 2: 30
Element 3: 40
Element 4: 50
```

Pointer to an Array Pointer:-

A pointer to an array pointer (often referred to as a double pointer) is a pointer that holds the address of another pointer, which in turn points to an array. This can be particularly useful when dealing with dynamic multi-dimensional arrays or when you want to modify an array of pointers.

1. Understanding Pointer to an Array Pointer

In C, you can create a pointer to an array pointer by declaring it as `type **pointer_name`. This pointer can point to the first element of an array of pointers.

2. Example: Pointer to an Array Pointer

Let’s see an example where we create an array of pointers to integers and then use a pointer to access and manipulate this array.

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

int main() {
    // Number of rows and columns
    int rows = 3;
    int cols = 4;

    // Allocate memory for an array of pointers (each pointer will point to an
array of integers)
    int **array = (int **)malloc(rows * sizeof(int *));
    for (int i = 0; i < rows; i++) {
        array[i] = (int *)malloc(cols * sizeof(int));
    }

    // Populate the array with values
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            array[i][j] = (i + 1) * (j + 1); // Example assignment
        }
    }

    // Pointer to the array of pointers
    int **ptr = array; // ptr points to the first element of the array of
pointers

    // Access and print the elements using the pointer
    printf("Array elements:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", *(ptr[i] + j)); // Accessing elements using pointer
arithmetic
        }
        printf("\n");
    }

    // Free the allocated memory
    for (int i = 0; i < rows; i++) {
        free(array[i]); // Free each row
    }
    free(array); // Free the array of pointers

    return 0;
}
```

3. Explanation of the Example

- **Dynamic Memory Allocation**:
    - We first allocate memory for an array of pointers, where each pointer will point to a row of integers.
    - We then allocate memory for each row to hold the actual integer values.
- **Pointer to Array Pointer**:
    - `int **ptr = array;` sets `ptr` to point to the first element of the `array`, which is the first pointer in the array of pointers.
- **Accessing Elements**:
    - We use pointer arithmetic to access the elements of the 2D array through the pointer to the array pointer. The expression `*(ptr[i] + j)` accesses the element at row `i` and column `j`.
- **Memory Cleanup**:
    - It's essential to free the memory allocated for each row and then for the array of pointers to avoid memory leaks.

4. Key Points

- **Double Pointer Syntax**: Use `type **pointer` to declare a pointer to an array pointer.
- **Dynamic Multi-Dimensional Arrays**: Double pointers are commonly used for creating dynamic multi-dimensional arrays in C.
- **Pointer Arithmetic**: You can use pointer arithmetic to access elements in the multi-dimensional array through the pointer to the array pointer.

3. Pointer and Address Arithmetic

Pointers support arithmetic operations to navigate through memory locations. Pointer arithmetic considers the data type it points to, meaning `p++` for an integer pointer moves the pointer by `sizeof(int)` bytes.

Examples:

```
int arr[3] = {1, 2, 3};
int *p = arr;

printf("%d\n", *p);   // Outputs: 1
p++;                   // Moves to the next element
printf("%d\n", *p);   // Outputs: 2
```

Valid Pointer Arithmetic Operations:

- `p++` or `p--`: Move to the next or previous memory location.
- `p + n` or `p - n`: Move `n` positions forward or backward.

4. Array Manipulation Using Pointers

Arrays and pointers are closely related. In fact, the name of an array represents a pointer to its first element.

Accessing Array Elements with Pointers:

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;

for (int i = 0; i < 5; i++) {
    printf("%d ", *(p + i)); // Access elements using pointer arithmetic
}
```

Using Pointers to Modify Array Elements:

```
int arr[3] = {10, 20, 30};
int *p = arr;
*p = 100; // Modifies arr[0]
*(p + 1) = 200; // Modifies arr[1]
```

Pointer to a Pointer:-

A **pointer to a pointer** (also known as a double pointer) is a type of pointer that stores the address of another pointer. This allows for multiple levels of indirection and can be particularly useful in dynamic memory allocation, multi-dimensional arrays, and certain data structures like linked lists and trees.

1. Declaration of Pointer to Pointer

A pointer to a pointer is declared using two asterisks (`**`). For example:

```
int **ptr; // ptr is a pointer to a pointer to an integer
```

2. Example: Pointer to Pointer

Here's an example to demonstrate how pointers to pointers work.

```
#include <stdio.h>

int main() {
    int value = 42;             // A normal integer variable
    int *ptr1 = &value;        // ptr1 is a pointer to value
    int **ptr2 = &ptr1;        // ptr2 is a pointer to ptr1

    printf("Value: %d\n", value);            // Outputs: 42
    printf("Value via ptr1: %d\n", *ptr1);   // Outputs: 42
    printf("Value via ptr2: %d\n", **ptr2);  // Outputs: 42

    // Modifying the value using pointer to pointer
    **ptr2 = 100; // Changes the value of 'value' to 100

    printf("Modified Value: %d\n", value); // Outputs: 100

    return 0;
}
```

3. Explanation of the Example

- `value` is an integer variable initialized to `42`.
- `ptr1` is a pointer to `value`, holding its address.
- `ptr2` is a pointer to `ptr1`, holding the address of `ptr1`.
- By dereferencing `ptr2` twice (`**ptr2`), we access the value of `value`.
- When we modify the value using `**ptr2`, it directly changes the original variable `value`.

4. Use Cases for Pointer to Pointer

1. **Dynamic Memory Allocation for Multi-Dimensional Arrays**: When you want to create a dynamic two-dimensional array.

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

   int main() {
       int rows = 3, cols = 4;
       int **arr = (int **)malloc(rows * sizeof(int *)); // Allocate memory
   for rows

       for (int i = 0; i < rows; i++) {
           arr[i] = (int *)malloc(cols * sizeof(int)); // Allocate memory for
   each column
       }
   ```

```
    // Assign values
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            arr[i][j] = i + j; // Example assignment
        }
    }

    // Print values
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    // Free allocated memory
    for (int i = 0; i < rows; i++) {
        free(arr[i]); // Free each row
    }
    free(arr); // Free the row pointers

    return 0;
}
```

Output:

```
0 1 2 3
1 2 3 4
2 3 4 5
```

In this example:

- We allocate a 2D array using pointers to pointers.
- Each row of the array is dynamically allocated.

2. **Managing Complex Data Structures**: Pointers to pointers can be useful in implementing complex data structures like linked lists, trees, or graphs where nodes may need to change their connections dynamically.

5. Key Points

- A pointer to a pointer allows multiple levels of indirection, which is useful in specific contexts.
- When using pointers to pointers, it’s essential to ensure that the memory they point to is allocated correctly to avoid dereferencing uninitialized pointers.
- It can simplify memory management for complex data structures and multi-dimensional arrays.

5. User-Defined Data Types: Structures and Unions

Structures and **unions** are user-defined data types that group different data types into a single unit. They are especially useful when working with records or objects with multiple properties.

Structures

A **structure** is a collection of variables (fields) of different data types grouped together under a single name. Structures are useful for modeling real-world entities.

Declaring and Using Structures:

```
#include <stdio.h>

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

int main() {
    struct Person person1 = {"John Doe", 30, 50000.50};

    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    printf("Salary: %.2f\n", person1.salary);
    return 0;
}
```

Accessing Structure Members with Pointers:

```
struct Person person2 = {"Jane Doe", 25, 60000.75};
struct Person *p = &person2;

printf("Name: %s\n", p->name); // Access using pointer
printf("Age: %d\n", p->age);
printf("Salary: %.2f\n", p->salary);
```

Unions

A **union** is similar to a structure but allows storing different data types in the same memory location. Only one member can store a value at a time, and modifying one member will affect others.

Declaring and Using Unions:

```
#include <stdio.h>

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

int main() {
    union Data data;

    data.i = 10;
    printf("i: %d\n", data.i);

    data.f = 220.5; // Overwrites the value of `i`
    printf("f: %.2f\n", data.f);

    sprintf(data.str, "Hello"); // Overwrites the values of `i` and `f`
    printf("str: %s\n", data.str);
    return 0;
}
```

Key Differences Between Structures and Unions:

- **Memory Allocation**: In a structure, memory is allocated for all members, whereas, in a union, memory is shared by all members.
- **Size**: The size of a structure is the sum of the sizes of its members, while the size of a union is equal to the size of its largest member.

1. Nested Structures

A **nested structure** is a structure defined within another structure. It is commonly used to represent hierarchical or multi-level data.

Example: Nested Structure

```
#include <stdio.h>

struct Address {
    char street[50];
    char city[50];
    int zipCode;
};

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

int main() {
    struct Person person = {
        "John Doe",
        30,
        {"123 Elm St", "Springfield", 12345}
    };

    printf("Name: %s\n", person.name);
    printf("Age: %d\n", person.age);
    printf("Address: %s, %s, %d\n", person.address.street, person.address.city,
person.address.zipCode);

    return 0;
}
```

Output:

```
Name: John Doe
Age: 30
Address: 123 Elm St, Springfield, 12345
```

In this example:

- `Address` is a structure nested within the `Person` structure.
- The `address` field in `Person` can store complete address information using the `Address` structure.

2. Nested Unions

A **nested union** is a union defined within a structure or another union. It’s useful when only one of several possible values needs to be stored at any given time.

Example: Nested Union within a Structure

```
#include <stdio.h>

struct Data {
    char type;
    union {
        int intValue;
        float floatValue;
        char strValue[20];
    } value;  // Nested union
};

int main() {
    struct Data data;

    // Assign an integer value
    data.type = 'i';
    data.value.intValue = 100;
    printf("Integer: %d\n", data.value.intValue);

    // Assign a float value
    data.type = 'f';
    data.value.floatValue = 220.5;
    printf("Float: %.2f\n", data.value.floatValue);

    // Assign a string value
    data.type = 's';
    snprintf(data.value.strValue, 20, "Hello");
    printf("String: %s\n", data.value.strValue);

    return 0;
}
```

Output:

```
Integer: 100
Float: 220.50
String: Hello
```

In this example:

- `Data` is a structure with a nested union called `value`.
- The union allows storing only one of `intValue`, `floatValue`, or `strValue` at a time.
- The `type` field indicates the type of data currently stored in the union.

3. Nested Structure and Union Together

You can nest structures and unions together to create even more complex data models.

Example: Structure with Nested Union and Structure

```
#include <stdio.h>

struct Date {
    int day;
```

```c
    int month;
    int year;
};

struct Record {
    int id;
    struct Date date;  // Nested structure
    union {
        int score;
        char grade;
    } result;  // Nested union
};

int main() {
    struct Record record1;

    // Set ID and Date
    record1.id = 1;
    record1.date.day = 15;
    record1.date.month = 8;
    record1.date.year = 2024;

    // Set result as a score
    record1.result.score = 95;
    printf("Record ID: %d\n", record1.id);
    printf("Date: %02d/%02d/%d\n", record1.date.day, record1.date.month, 
record1.date.year);
    printf("Score: %d\n", record1.result.score);

    // Set result as a grade
    record1.result.grade = 'A';
    printf("Grade: %c\n", record1.result.grade);

    return 0;
}
```

Output:

```
Record ID: 1
Date: 15/08/2024
Score: 95
Grade: A
```

In this example:

- `Record` is a structure containing both a nested structure (`Date`) and a nested union (`result`).
- The nested `Date` structure stores the date of the record.
- The nested `result` union allows `Record` to store either a score or a grade, but not both simultaneously.

Key Points

- **Nested Structures**: Use when you need to organize data hierarchically.
- **Nested Unions**: Use when only one of several fields will hold a value at any one time, helping save memory.
- **Combination of Structures and Unions**: Combine them as needed to create complex and efficient data models.

Structures with Pointers

A structure can contain pointers as its members, allowing it to point to other data or even other structures. This is useful for creating linked data structures, such as linked lists or trees.

Example: Structure with Pointer Members

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

// Define a structure with a pointer member
struct Person {
    char *name;
    int age;
};

int main() {
    // Create an instance of the structure
    struct Person person;

    // Allocate memory for the name pointer
    person.name = (char *)malloc(20 * sizeof(char));

    // Assign values to structure members
    if (person.name != NULL) {
        snprintf(person.name, 20, "Alice");
    }
    person.age = 25;

    // Display the structure contents
    printf("Name: %s\n", person.name);
    printf("Age: %d\n", person.age);

    // Free allocated memory
    free(person.name);

    return 0;
}
```

Explanation:

1. **Pointer Member**: The `name` member in `Person` is a `char*`, which allows for dynamic memory allocation.
2. **Memory Allocation**: Memory is allocated for `name` using `malloc`.
3. **Accessing Members**: You access `name` and `age` just as you would with any structure member.
4. **Freeing Memory**: After usage, memory allocated with `malloc` is freed to prevent memory leaks.

Expected Output:

```
Name: Alice
Age: 25
```

2. Pointer to Structure

Pointers can also point to structures themselves. This is useful when you want to pass large structures to functions without copying them, as pointers only store the address.

Example: Pointer to Structure

```
#include <stdio.h>

// Define a structure
struct Person {
    char name[20];
    int age;
};

// Function to display structure details
void displayPerson(struct Person *p) {
    printf("Name: %s\n", p->name);
    printf("Age: %d\n", p->age);
}

int main() {
    struct Person person = {"Bob", 30}; // Initialize a structure
    struct Person *ptr = &person;       // Pointer to the structure

    // Use the pointer to display the structure's contents
    displayPerson(ptr);

    return 0;
}
```

Explanation:

1. **Pointer to Structure**: `ptr` is a pointer to the structure `person`.
2. **Arrow Operator (`->`)**: When using a pointer to access structure members, the `->` operator is used (e.g., `p->name` and `p->age`).

Expected Output:

```
makefile
Copy code
Name: Bob
Age: 30
```

---

3. Pointer to Pointer (Double Pointer)

A pointer to a pointer (or double pointer) stores the address of another pointer. This is common in dynamic memory management, multi-dimensional arrays, and when working with pointers to dynamically allocated structures.

Example: Pointer to Pointer

```
#include <stdio.h>

int main() {
    int value = 10;
    int *ptr1 = &value;  // Pointer to an int
    int **ptr2 = &ptr1;  // Pointer to pointer

    printf("Value: %d\n", value);       // Original value
    printf("Value using ptr1: %d\n", *ptr1);   // Dereferencing ptr1
    printf("Value using ptr2: %d\n", **ptr2);  // Dereferencing ptr2 twice

    return 0;
}
```

Explanation:

1. **Pointer to Pointer**: `ptr2` is a pointer to `ptr1`, which in turn points to `value`.
2. **Double Dereference**: `**ptr2` accesses the original `value` by dereferencing twice.

Expected Output:

```
Value: 10
Value using ptr1: 10
Value using ptr2: 10
```

---

4. Array of Pointers (Pointer to Pointer Example with Strings)

An array of pointers can also be useful, especially when dealing with a collection of strings or dynamically allocated arrays.

Example: Array of Pointers to Strings

```
#include <stdio.h>

int main() {
    const char *names[] = {"Alice", "Bob", "Charlie"}; // Array of pointers to
strings
    int count = sizeof(names) / sizeof(names[0]);

    for (int i = 0; i < count; i++) {
        printf("Name %d: %s\n", i + 1, names[i]);
    }

    return 0;
}
```

Explanation:

1. **Array of Pointers**: `names` is an array of pointers, where each pointer points to a string literal.
2. **Pointer to Pointer**: Effectively, `names` acts like a `char**`, where each element points to a `char*`.

Expected Output:

```
Name 1: Alice
Name 2: Bob
Name 3: Charlie
```

Structure pointer variable:

In C, a **struct pointer variable** is a pointer that points to a structure. This pointer variable allows you to work with the memory address of a structure rather than copying the structure's entire data, which can improve memory efficiency when passing structures to functions or manipulating large structures.

Here's how you can use a struct pointer variable, including how to declare, initialize, and access the structure's members using a pointer.

Basic Example: Using a Struct Pointer Variable

Let's walk through an example of defining a struct, creating a pointer to it, and accessing its members.

Step 1: Define a Structure

```
#include <stdio.h>

// Define a structure
struct Person {
    char name[50];
    int age;
};
```

In this example, `Person` is a structure with two members:

- `name`: a character array to store the person's name.
- `age`: an integer to store the person's age.

Step 2: Declare and Initialize a Struct Pointer Variable

To create a pointer variable for the structure, use the following syntax:

```
int main() {
    struct Person person1 = {"Alice", 30};   // Create an instance of the
structure
    struct Person *ptr = &person1;           // Pointer to the structure

    // Access structure members using the pointer
    printf("Name: %s\n", ptr->name);
    printf("Age: %d\n", ptr->age);

    return 0;
}
```

Explanation of the Code:

1. **Pointer Declaration**: `struct Person *ptr` declares a pointer variable, `ptr`, of type `struct Person*`.
2. **Pointer Initialization**: `ptr = &person1;` initializes the pointer to point to the `person1` structure.
3. **Accessing Members via Pointer**:

- To access structure members via a pointer, use the `->` (arrow) operator.
- `ptr->name` accesses the `name` member of the structure that `ptr` points to.
- `ptr->age` accesses the `age` member.

Expected Output:

```
Name: Alice
Age: 30
```

Example: Using a Struct Pointer in a Function

Struct pointer variables are particularly useful when passing structures to functions. Passing a pointer instead of the whole structure makes the function more efficient because only the address of the structure is passed, not a copy of the entire structure.

```
#include <stdio.h>

// Define a structure
struct Person {
    char name[50];
    int age;
};

// Function that takes a pointer to a structure
void displayPerson(struct Person *p) {
    printf("Name: %s\n", p->name);
    printf("Age: %d\n", p->age);
}

int main() {
    struct Person person1 = {"Bob", 28}; // Initialize a structure
    displayPerson(&person1);             // Pass the address of person1 to the
function

    return 0;
}
```

Explanation of the Code:

1. **Function with Struct Pointer Parameter**: `displayPerson` takes a `struct Person*` as an argument. This allows it to access the original structure without making a copy.
2. **Calling the Function**: `displayPerson(&person1);` passes the address of `person1` to the function.
3. **Accessing Members in the Function**: Inside `displayPerson`, `p->name` and `p->age` access the structure members using the pointer.

Expected Output:

```
Name: Bob
Age: 28
```

Example: Dynamic Memory Allocation with Struct Pointer Variables

You can use a struct pointer variable to allocate memory for a structure dynamically using `malloc`.

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

// Define a structure
struct Person {
    char name[50];
    int age;
};

int main() {
    // Dynamically allocate memory for a structure
    struct Person *ptr = (struct Person *)malloc(sizeof(struct Person));
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Initialize structure members via pointer
    snprintf(ptr->name, sizeof(ptr->name), "Charlie");
    ptr->age = 35;

    // Display structure members
    printf("Name: %s\n", ptr->name);
    printf("Age: %d\n", ptr->age);

    // Free allocated memory
    free(ptr);

    return 0;
}
```

Explanation of the Code:

1. **Memory Allocation**: `malloc` allocates memory for the structure and returns a pointer to it.
2. **Member Initialization**: `ptr->name` and `ptr->age` are set directly via the pointer.
3. **Freeing Memory**: After using the dynamically allocated structure, `free(ptr);` releases the memory.

Expected Output:

```
Name: Charlie
Age: 35
```

Summary

- **Structure Pointer Variable**: A pointer that points to a structure, allowing efficient manipulation and passing of structure data.
- **Arrow Operator (`->`)**: Used to access members of a structure through a pointer.
- **Efficient Passing to Functions**: Passing a structure pointer to functions avoids copying the structure data.

- **Dynamic Memory Allocation**: Allows creating structures on the heap, which are accessed via pointers.

Union pointer variables:-

In C, **union pointer variables** are pointers that point to a union. Just like with structures, using a pointer to a union allows you to work with the memory address of the union rather than copying the entire union, which can be useful when dealing with larger or frequently used unions.

Let's go through some examples on how to define unions, create pointers to unions, and access union members via pointers.

Defining a Union and Using a Pointer to the Union

To create a pointer to a union, you:

1. Define the union.
2. Declare a pointer to that union type.
3. Access union members through the pointer.

Example 1: Basic Union with Pointer

```
#include <stdio.h>

// Define a union
union Data {
    int intValue;
    float floatValue;
    char charValue;
};

int main() {
    union Data data;         // Declare a union variable
    union Data *ptr = &data; // Declare a pointer to the union and initialize it

    // Access and modify union members via the pointer
    ptr->intValue = 42;      // Set intValue
    printf("Integer: %d\n", ptr->intValue);

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

    ptr->charValue = 'A';    // Set charValue (overwrites floatValue)
    printf("Char: %c\n", ptr->charValue);

    return 0;
}
```

Explanation:

1. **Union Definition**: `union Data` has three members: `intValue`, `floatValue`, and `charValue`. Since a union shares memory for all its members, assigning one member overwrites the others.
2. **Pointer to Union**: `union Data *ptr = &data;` creates a pointer to the union.

3. **Accessing Members**: Use the `->` operator to access members of the union via the pointer.

Expected Output:

```
Integer: 42
Float: 3.14
Char: A
```

Note that each assignment overwrites the previous value since all members share the same memory.

Example 2: Using Dynamic Memory Allocation with a Union Pointer

Using dynamic memory allocation with unions can be useful when you need to allocate memory for a union at runtime.

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

// Define a union
union Data {
    int intValue;
    float floatValue;
    char charValue;
};

int main() {
    // Allocate memory for the union using malloc
    union Data *ptr = (union Data *)malloc(sizeof(union Data));
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Set and access members via pointer
    ptr->intValue = 100;
    printf("Integer: %d\n", ptr->intValue);

    ptr->floatValue = 9.81f;  // Overwrites intValue
    printf("Float: %.2f\n", ptr->floatValue);

    ptr->charValue = 'Z';     // Overwrites floatValue
    printf("Char: %c\n", ptr->charValue);

    // Free allocated memory
    free(ptr);

    return 0;
}
```

Explanation:

1. **Dynamic Memory Allocation**: `malloc` is used to allocate memory for the union, and `ptr` points to this dynamically allocated memory.
2. **Accessing Members**: `ptr->intValue`, `ptr->floatValue`, and `ptr->charValue` are accessed and modified using the pointer.

3. **Memory Management**: `free(ptr);` releases the allocated memory after it is no longer needed.

Expected Output:

```
Integer: 100
Float: 9.81
Char: Z
```

Example 3: Passing a Union Pointer to a Function

You can also pass a pointer to a union as a function parameter. This approach avoids copying the union, allowing the function to access or modify the original union data.

```
#include <stdio.h>

// Define a union
union Data {
    int intValue;
    float floatValue;
    char charValue;
};

// Function to display union values using a union pointer
void displayUnion(union Data *dataPtr) {
    printf("Integer: %d\n", dataPtr->intValue);
    printf("Float: %.2f\n", dataPtr->floatValue);
    printf("Char: %c\n", dataPtr->charValue);
}

int main() {
    union Data data;
    union Data *ptr = &data;  // Pointer to the union

    ptr->intValue = 25;       // Assign to intValue
    displayUnion(ptr);

    ptr->floatValue = 7.89;   // Overwrite with floatValue
    displayUnion(ptr);

    ptr->charValue = 'X';     // Overwrite with charValue
    displayUnion(ptr);

    return 0;
}
```

Explanation:

1. **Function Parameter**: `displayUnion` takes a `union Data*` pointer, allowing it to access and print the union's members.
2. **Passing Union Pointer**: `displayUnion(ptr);` passes the pointer to the function, which then uses `dataPtr->member` to access union members.

Expected Output:

```
Integer: 25
```

```
Float: 7.89
Char: X
```

Each call to `displayUnion` shows the latest value assigned, illustrating how union members overwrite each other.

Summary

- **Union Pointer Variable**: A pointer that points to a union, allowing access to union members via the pointer.
- **Arrow Operator (`->`)**: Used to access members of a union through a pointer.
- **Dynamic Memory**: `malloc` can allocate memory for unions, and the pointer accesses this memory.
- **Function Parameters**: Passing union pointers to functions enables efficient access to union data without copying.

Structure or union with Function Pointers:-

In C, structures cannot contain function members directly as they do in some object-oriented languages like C++. However, you can achieve similar behavior by using function pointers as members within a structure. This approach allows you to store pointers to functions that can be invoked through the structure, creating a similar effect to methods in classes.

Here's how to define a structure with function pointers and call these functions via the structure.

Example: Structure with Function Pointer as "Function Member"

Let's define a structure to represent a `Rectangle` with function pointers for calculating its area and perimeter.

```
#include <stdio.h>

// Define a structure with function pointers
struct Rectangle {
    int length;
    int width;

    // Function pointers for area and perimeter
    int (*area)(int, int);
    int (*perimeter)(int, int);
};

// Function to calculate area
int calculateArea(int length, int width) {
    return length * width;
}

// Function to calculate perimeter
int calculatePerimeter(int length, int width) {
    return 2 * (length + width);
}

int main() {
    // Create an instance of Rectangle
    struct Rectangle rect;
    rect.length = 10;
    rect.width = 5;

    // Assign function pointers
    rect.area = calculateArea;
    rect.perimeter = calculatePerimeter;

    // Use function pointers through the structure
    printf("Area: %d\n", rect.area(rect.length, rect.width));
    printf("Perimeter: %d\n", rect.perimeter(rect.length, rect.width));

    return 0;
}
```

Explanation of the Code

1. **Structure Definition**:
    - `struct Rectangle` contains `length` and `width` as data members.
    - It also has two function pointers, `area` and `perimeter`, which point to functions that take two integers (length and width) and return an integer.
2. **Function Definitions**:
    - `calculateArea` calculates the area of the rectangle.
    - `calculatePerimeter` calculates the perimeter of the rectangle.
3. **Assigning Function Pointers**:
    - `rect.area = calculateArea;` assigns the `calculateArea` function to the `area` function pointer within `rect`.
    - `rect.perimeter = calculatePerimeter;` assigns the `calculatePerimeter` function to the `perimeter` function pointer.
4. **Using Function Pointers**:
    - `rect.area(rect.length, rect.width)` calls the `calculateArea` function via the `area` function pointer.
    - `rect.perimeter(rect.length, rect.width)` calls the `calculatePerimeter` function via the `perimeter` function pointer.

Expected Output

```
Area: 50
Perimeter: 30
```

Explanation of the Output

- The area is calculated as `length * width = 10 * 5 = 50`.
- The perimeter is calculated as `2 * (length + width) = 2 * (10 + 5) = 30`.

Additional Example: Using Multiple Function Pointers in a Structure

Here’s a more generic example with a structure that has multiple function pointers for performing different arithmetic operations.

```
#include <stdio.h>

// Define a structure with function pointers for basic arithmetic operations
struct Calculator {
    int (*add)(int, int);
    int (*subtract)(int, int);
    int (*multiply)(int, int);
    float (*divide)(int, int);
};

// Functions for basic arithmetic operations
int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int multiply(int a, int b) {
    return a * b;
}

float divide(int a, int b) {
    if (b != 0) {
        return (float)a / b;
    } else {
        printf("Error: Division by zero\n");
        return 0.0;
    }
}

int main() {
    // Create an instance of Calculator
    struct Calculator calc;

    // Assign function pointers
    calc.add = add;
    calc.subtract = subtract;
    calc.multiply = multiply;
    calc.divide = divide;

    // Use function pointers through the structure
    int x = 20, y = 10;
    printf("Add: %d\n", calc.add(x, y));
    printf("Subtract: %d\n", calc.subtract(x, y));
    printf("Multiply: %d\n", calc.multiply(x, y));
    printf("Divide: %.2f\n", calc.divide(x, y));

    return 0;
}
```

Explanation of the Code

1. **Structure with Function Pointers**: `struct Calculator` has four function pointers to add, subtract, multiply, and divide integers.

2. **Function Assignments**: Each function pointer in `calc` is assigned to a corresponding arithmetic function.
3. **Using the Structure**: We call each operation (add, subtract, multiply, divide) through the `calc` structure.

Expected Output

```
Add: 30
Subtract: 10
Multiply: 200
Divide: 2.00
```

Summary

Using function pointers in structures provides a way to associate specific behaviors (functions) with data, similar to methods in an object-oriented programming language. This is a powerful technique to implement function-like behavior in C structures, useful in modular code design and simulations of object-oriented behavior in C.

UNIT V: Function & File Handling

Introduction to Functions, Function Declaration and Definition, Function call Return Types and Arguments, modifying parameters inside functions using pointers, arrays as parameters. Scope and Lifetime of Variables, Basics of File Handling.

Introduction to Functions

A **function** is a self-contained block of code designed to perform a specific task. Functions promote code reusability, readability, and modular programming by allowing developers to separate logic into smaller, manageable units. Functions are a core part of most programming languages and are particularly important in C.

1. Function Declaration and Definition

- **Function Declaration** (or **Prototype**): The declaration of a function specifies its name, return type, and parameters, telling the compiler about the function's existence.

  ```
  int add(int a, int b); // Declaration: tells the compiler that the
  function "add" exists.
  ```

- **Function Definition**: The definition provides the actual implementation of the function.

  ```
  int add(int a, int b) { // Definition
      return a + b;
  }
  ```

2. Function Call

A **function call** is the process of invoking a function, usually by passing arguments to it. Functions can be called from other functions (including main) to perform their defined tasks.

```
int result = add(5, 3); // Calls the add function with arguments 5 and 3
```

3. Return Types and Arguments

- **Return Type**: A function's return type specifies the type of value it will return. Common return types include `int`, `float`, `void` (no return value), etc.

  ```
  float calculateArea(float radius) {
      return 3.14 * radius * radius; // Returns a float value
  }
  ```

- **Arguments (Parameters)**: Parameters allow data to be passed into a function. A function can have any number of parameters or none at all.

4. Modifying Parameters Inside Functions Using Pointers

To modify the values of parameters inside a function, we can pass pointers to those parameters instead of passing them by value.

Example:

```
void updateValue(int *p) { // Function takes a pointer as a parameter
    *p = 20; // Modifies the original variable's value
}

int main() {
    int num = 10;
    updateValue(&num); // Pass the address of num
    printf("%d", num); // Output: 20
    return 0;
}
```

5. Arrays as Parameters

When passing an array to a function, what is actually passed is a pointer to the array's first element, allowing the function to access and modify the array's contents.

Example:

```
void modifyArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] *= 2; // Modify each element
    }
}

int main() {
    int numbers[3] = {1, 2, 3};
    modifyArray(numbers, 3);
    for (int i = 0; i < 3; i++) {
        printf("%d ", numbers[i]); // Output: 2 4 6
    }
    return 0;
}
```

Returning an Array as a Pointer:-

Example Code:

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

// Function that returns a pointer to an integer array
int* createArray(int size) {
    int *arr = (int *)malloc(size * sizeof(int)); // Allocate memory for the
array
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    // Initialize the array
    for (int i = 0; i < size; i++) {
        arr[i] = i * 2; // Example initialization
    }

    return arr; // Return pointer to the array
}

int main() {
    int size = 5;
    int *array = createArray(size); // Get the pointer to the array

    // Print the array
    printf("Returned Array:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    free(array); // Free the allocated memory

    return 0;
}
```

Expected Output:

```
Returned Array:
0 2 4 6 8
```

2. Passing an Array as a Pointer Parameter

Example Code:

```
#include <stdio.h>

// Function that takes a pointer to an integer array and its size
void modifyArray(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] += 5; // Modify each element by adding 5
    }
}

int main() {
    int array[] = {1, 2, 3, 4, 5};
    int size = sizeof(array) / sizeof(array[0]);
    printf("Original Array:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    modifyArray(array, size); // Pass the array to the function

    printf("Modified Array:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

Expected Output:

```
Original Array:
1 2 3 4 5
Modified Array:
6 7 8 9 10
```

----

6. Scope and Lifetime of Variables

- **Scope**: Defines where a variable can be accessed. It can be:
    - **Local Scope**: Variables declared within a function have local scope, meaning they are accessible only within that function.
    - **Global Scope**: Variables declared outside of all functions are global and accessible throughout the program.
- **Lifetime**: Defines the duration for which a variable retains its value.
    - **Automatic (Local) Variables**: Exist only within the function they are declared in and are destroyed when the function exits.
    - **Static Variables**: Retain their value between function calls and are initialized only once.
    - **Global Variables**: Exist for the entire duration of the program.

Example of Scope and Lifetime:

```
#include <stdio.h>

int globalVar = 5; // Global variable

void demoFunction() {
    static int staticVar = 0; // Static variable
    int localVar = 10; // Local variable
    staticVar++;
    printf("Static: %d, Local: %d\n", staticVar, localVar);
}

int main() {
    demoFunction(); // Output: Static: 1, Local: 10
    demoFunction(); // Output: Static: 2, Local: 10
    return 0;
}
```

7. Basics of File Handling

File handling allows a program to store data persistently on the file system. C provides several functions for working with files, allowing us to open, read, write, and close files.

```
File handling in C allows you to read from and write to files on disk. This capability is essential for data persistence, enabling your programs to store and retrieve data beyond their runtime. The C standard library provides several functions for file operations.
```

Basic File Operations

The primary operations you can perform on files are:

1. **Opening a file**
2. **Reading from a file**
3. **Writing to a file**
4. **Closing a file**

File Modes

When opening a file, you specify the mode of operation using a string:

- `"r"`: Read mode (file must exist).
- `"w"`: Write mode (creates a new file or truncates an existing one).
- `"a"`: Append mode (writes data at the end of the file).
- `"r+"`: Read and write mode (file must exist).
- `"w+"`: Read and write mode (creates a new file or truncates an existing one).
- `"a+"`: Read and append mode (writes data at the end of the file).

File Pointer

In C, files are handled using file pointers of type `FILE*`. You use the `fopen` function to open a file and get a file pointer, and `fclose` to close it.

Basic File Handling Functions

- `fopen()`: Opens a file and returns a file pointer.
- `fclose()`: Closes an open file.
- `fread()`: Reads data from a file.
- `fwrite()`: Writes data to a file.
- `fprintf()`: Writes formatted data to a file.
- `fscanf()`: Reads formatted data from a file.
- `fgets()`: Reads a string from a file.
- `fputs()`: Writes a string to a file.
- `feof()`: Checks if the end of the file has been reached.

Example: File Writing and Reading

Here is a complete example demonstrating file handling in C. It writes some text to a file and then reads it back.

Basic File Operations:

1. **Opening a File**: `fopen` is used to open a file. It requires the file name and mode (`"r"`, `"w"`, `"a"`, etc.).

   ```
   FILE *filePtr = fopen("example.txt", "w"); // Open file in write mode
   ```

2. **Writing to a File**: Use `fprintf` or `fputs` for writing data.

   ```
   fprintf(filePtr, "Hello, World!\n");
   ```

3. **Reading from a File**: Use `fscanf`, `fgets`, or `fgetc` for reading data.

   ```
   char line[50];
   fgets(line, 50, filePtr); // Read a line from the file
   ```

4. **Closing a File**: `fclose` closes the file, freeing resources.

   ```
   fclose(filePtr);
   ```

Example Code:

```
#include <stdio.h>

int main() {
    FILE *file; // Declare a file pointer
    char data[100];

    // Writing to a file
    file = fopen("example.txt", "w"); // Open file in write mode
    if (file == NULL) {
        printf("Error opening file for writing.\n");
        return 1; // Exit if file opening fails
    }

    fprintf(file, "Hello, World!\n"); // Write formatted data to the file
    fprintf(file, "This is a file handling example in C.\n");
    fclose(file); // Close the file after writing

    // Reading from a file
    file = fopen("example.txt", "r"); // Open file in read mode
    if (file == NULL) {
        printf("Error opening file for reading.\n");
        return 1; // Exit if file opening fails
    }

    printf("Contents of the file:\n");
    while (fgets(data, sizeof(data), file) != NULL) {
        printf("%s", data); // Read and print each line
    }

    fclose(file); // Close the file after reading

    return 0;
}
```

Explanation of the Example:

1. **Opening a File**:
    - The program opens a file named `example.txt` in write mode (`"w"`). If the file does not exist, it creates a new one.
    - It checks if the file was opened successfully by verifying that the file pointer is not `NULL`.
2. **Writing to the File**:
    - The program writes two lines of text to the file using `fprintf`.
3. **Closing the File**:
    - After writing, it closes the file using `fclose`.
4. **Reading from the File**:
    - The program then opens the same file in read mode (`"r"`).
    - It uses a loop with `fgets` to read each line from the file until it reaches the end of the file.
    - Each line is printed to the console.
5. **Final Cleanup**:
    - The file is closed again after reading.

Expected Output:

If you run the program, it will write the following lines to `example.txt`:

```
Hello, World!
This is a file handling example in C.
```

When reading the file, it will output:

```
Contents of the file:
Hello, World!
This is a file handling example in C.
```