Module-3
Arrays and Strings

School of Electronics Engineering (SENSE)
Vellore Institute of Technology
Chennai

Module 3: Arrays and Strings

Outline

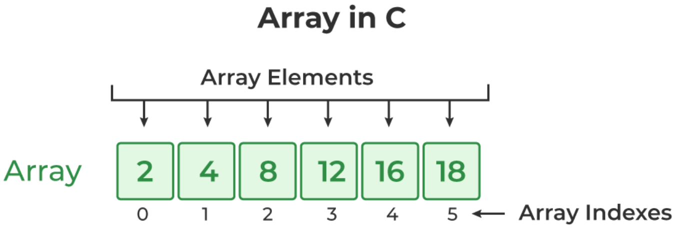
Introduction to Arrays

Definition

An array is a collection of items stored at contiguous memory locations. In C, arrays are used to store similar types of elements.

Application in Embedded Systems

Arrays are used in embedded systems for handling multiple similar data efficiently, such as sensor readings, buffer storage, and lookup tables.



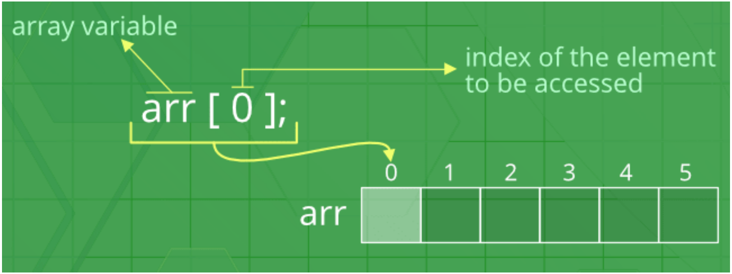
One-Dimensional Arrays

Syntax and Declaration

```
int arr[10]; // Declares an array of 10 integers
```

Example

```
arr[0] = 1; // Sets the first element to 1
```



Declaration of One-Dimensional Arrays

Array Declaration Syntax

```
type arrayName[arraySize];
```

Example: Sensor Readings Array

```
#define NUM_SENSORS 4
int sensorReadings[NUM_SENSORS]; //Array for storing sensor va
```

Note on Embedded Systems

In embedded C, the size of arrays is often determined by the number of physical components, like sensors or actuators, connected to the microcontroller.

Initializing One-Dimensional Arrays

Array Initialization Syntax

```
type arrayName[arraySize] = {val1, val2, ..., valN};
```

Example: Setting Initial Sensor States

```
int sensorStates[NUM_SENSORS] = {0};  // Initialize all to 0
```

Embedded Systems Context

Initialization is crucial in embedded systems to ensure that memory has defined values before use, particularly for registers or state variables.

Accessing Array Elements

Accessing Elements Syntax

Elements in an array are accessed using their index.

```
arrayName[index]
```

Example: Accessing an Element

```
int array[5] = {1, 2, 3, 4, 5};
int firstElement = array[0]; // Access first element
```

Embedded Systems Consideration

When accessing array elements in embedded systems, ensure that the index is within the bounds to prevent undefined behavior and potential system crashes.

Iterating Over Arrays

Iterating Over Arrays

To perform operations on each element in an array, a loop is used.

```
for (int i = 0; i < arraySize; i++) {
    // Code to execute
}
```

Example: Summing Array Elements

```
int sum = 0;
for (int i = 0; i < 5; i++) {
    sum += array[i];
}
```

Embedded Systems Tip

In time-critical embedded applications, consider the loop's impact on execution time and optimize the iteration process.

Example: Summing Elements in an Array

Standard C Example

```c
int main() {
    int values[5] = {5, 10, 15, 20, 25};
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        sum += values[i];
    }
    printf("Sum of values: %d\n", sum);
    return 0;
}
```

Multi-Dimensional Arrays Overview

Definition

Multi-dimensional arrays are arrays of arrays.
They are used to represent data in more than one dimension, such as matrices.

Multi-Dimensional Arrays

Syntax and Declaration

```
int multiArr[3][4]; // Declares a 3x4 array
```

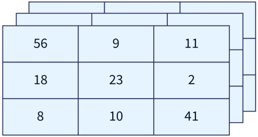
Example

```
multiArr[0][1] = 5; // Element at row 0, column 1 to 5
```

2-D Array

	Column 0	Column 1	Column 2
Row 0	a[0][0]	a[0][1]	a[0][2]
Row 1	a[1][0]	a[1][1]	a[1][2]
Row 2	a[2][0]	a[2][1]	a[2][2]
Row 3	a[3][0]	a[3][1]	a[3][2]
Row 4	a[4][0]	a[4][1]	a[4][2]

3-D Array



Declaration of Multi-Dimensional Arrays

Declaration Syntax

```
type arrayName[size1][size2];
```

Example: 2D Array for LED Matrix

```
#define ROWS 3
#define COLS 3
int ledMatrix[ROWS][COLS];  // LED states for a 3x3 matrix
```

Embedded C Context

Such arrays can represent physical layouts in hardware, like an LED matrix, with each element controlling the state of an LED.

Initializing Multi-Dimensional Arrays

Initialization Syntax

```
type arrayName[size1][size2] = {{val1, val2}, {...}};
```

Standard C Example

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

Embedded C Application

Initializing state matrices for devices like displays where each element represents a pixel or segment state.

Accessing Multi-Dimensional Array Elements

Accessing Elements

Use row and column indices to access elements in a multi-dimensional array.

```
arrayName[row][column]
```

Standard C Example

```
int value = matrix[1][2]; // Accesses the element at second ro
```

Embedded C Context

For embedded systems, ensure the indices are within bounds to maintain system stability.

Nested Loops and Multi-Dimensional Arrays

Using Nested Loops

Nested loops allow iteration over rows and columns of a multi-dimensional array.

```
for(int i = 0; i < rows; i++) {
    for(int j = 0; j < columns; j++) {
        // Access array elements
    }
}
```

Standard C Example

```
for(int i = 0; i < 2; i++) {
    for(int j = 0; j < 3; j++) {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}
```

Embedded C Consideration

In embedded systems, nested loops are commonly used for scanning or controlling a grid of sensors or actuators.

Example: Matrix Addition

Standard C Example - Adding Two Matrices

```
void addMatrices(int A[2][3], int B[2][3], int C[2][3]) {
    for(int i = 0; i < 2; i++) {
        for(int j = 0; j < 3; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

Embedded C Application

Matrix addition can be used in embedded systems for combining data from multiple sensor arrays.

Arrays in Memory: How C Stores Arrays

Memory Layout of Arrays

Discuss how arrays are contiguous blocks of memory and how multi-dimensional arrays are stored in row-major order.

Embedded C Significance

Understanding memory layout is crucial in embedded systems for optimizing data storage and access patterns.

```
int num[3][4] = {
  {1, 2,  3,  4},
  {5, 6,  7,  8},
  {9, 10, 11, 12}
};
```

row-wise memory allocation

	<— row 0 —>				<— row 1 —>				<— row 2 —>			
value	1	2	3	4	5	6	7	8	9	10	11	12
address	1000	1002	1004	1006	1008	1010	1012	1014	1016	1018	1020	1022

first element of the array num

Address Arithmetic in Arrays

Understanding Address Arithmetic

- Addresses of array elements are calculated using the base address and the size of the element type.
- This is essential for pointer arithmetic and understanding how arrays are accessed in memory.

Standard C Example

```
int array[5];
int *ptr = array;
printf("%p %p", ptr, ptr + 1);  // Prints contiguous addresses
```

Embedded C Application

Directly manipulating memory addresses is common in embedded systems, for instance when interfacing with hardware registers.

Example: Searching an Array

Implementing a Search Algorithm

- A linear search algorithm iterates over an array to find a value.
- This is a straightforward example of how to traverse an array with a loop.

Standard C Code for Linear Search

```
int linearSearch(int arr[], int size, int value) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == value) return i;
    }
    return -1;  // Value not found
}
```

Embedded C Scenario

Searching through a data array to find a sensor reading that exceeds a threshold could trigger an event or alert.

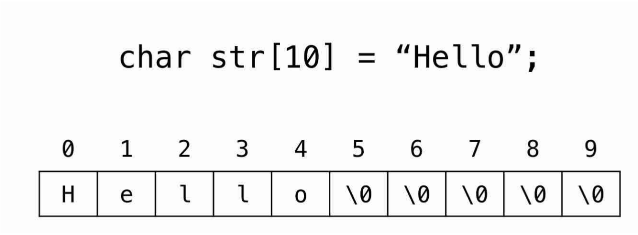
Strings in C: A Special Kind of Array

What Are Strings in C?

In C, strings are arrays of characters terminated by a null character \0.

Usage in Embedded Systems

Strings are often used for storing data read from or to be written to peripherals, like displays in embedded systems.



Declaring and Initializing Strings

Declaration and Initialization

```
char str[] = "Hello, World!";
```

Embedded C Example

```
char errorMessage[20] = "Error Code: ";
```

Note

String initialization automatically includes the null terminator.

Reading and Writing Strings

Using Standard I/O Functions

```
scanf("%s", str);
printf("%s", str);
```

Embedded C Considerations

In embedded systems, functions like ‘sprintf‘ and ‘sscanf‘ are used for formatting strings to interact with hardware or protocol messages.

String Manipulation Functions

Common Functions

- ‘strlen‘ - Get string length
- ‘strcpy‘ - Copy string
- ‘strcat‘ - Concatenate strings
- ‘strcmp‘ - Compare two strings

Embedded Systems Note

Use these functions carefully to avoid buffer overflows, which are critical in the context of embedded systems with limited memory.

Example: String Concatenation

Concatenating Two Strings

```
char greeting[50] = "Hello, ";
char name[] = "John";
strcat(greeting, name);
```

Embedded C Application

String concatenation might be used in embedded systems for creating log messages or protocol frames.

Functions in C

Definition and Purpose

Functions are reusable blocks of code that perform a specific task. They help modularize the code, making it more readable and maintainable.

Embedded Systems Context

Functions in embedded systems are used to encapsulate hardware control operations, algorithms, and routines.

Declaring and Defining Functions

Function Declaration (Prototype)

```
void functionName(parameters);
```

Function Definition

```
void functionName(parameters) {
    // Code to execute
}
```

Note

Function prototypes are often declared in header files, while definitions are in source files.

Declaring and Defining Functions

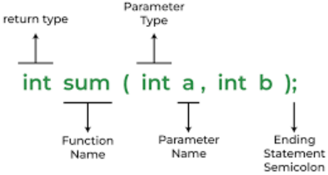
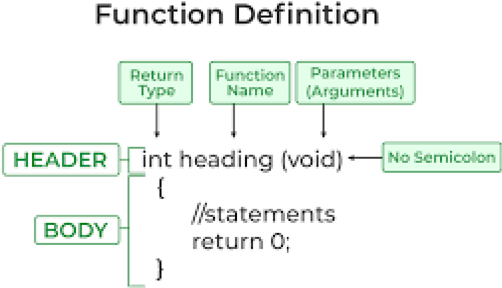


Figure: Function Declaration



Calling Functions in C

Calling a Function

```
functionName(arguments);
```

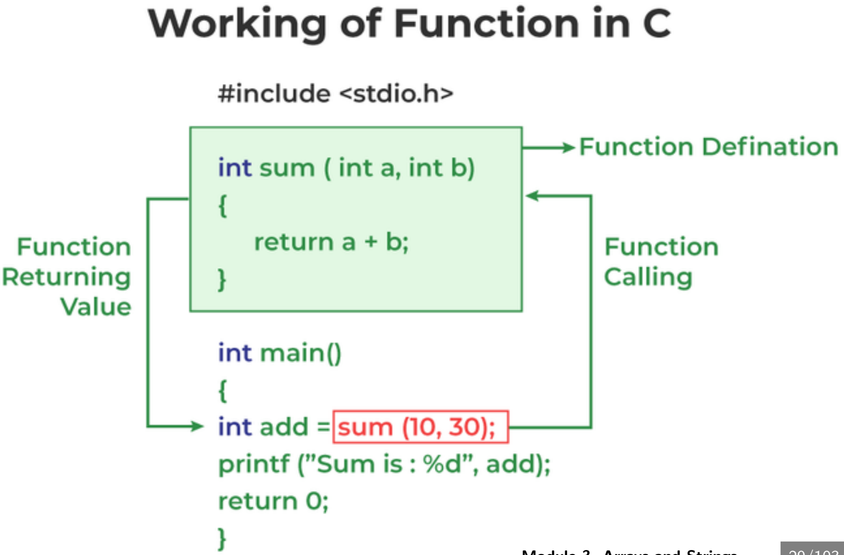
Example

```
void turnOnLED(int ledNumber);
turnOnLED(1); // Turns on LED number 1
```

Embedded C Tip

Ensure that any functions that interface with hardware are called with the correct timing and context to avoid system errors.

Calling Functions in C



Passing Parameters to Functions

Parameter Passing

In C, parameters can be passed by value, where a copy of the data is made, or by reference, using pointers, which allows the function to modify the original data.

Pass by Value Example

```c
void setTemperature(int temp);
```

Pass by Reference Example

```c
void resetCounter(int *counter) {
    *counter = 0;
}
```

The Return Statement and Return Types

Returning Values from Functions

Functions in C can return a value. The type of the return value must match the function's return type.

Return Statement Example

```
int getSensorData() {
    return sensorValue; // Assume sensorValue is an int
}
```

Embedded C Application

Functions that interact with hardware components often return status codes, data readings, or boolean values indicating success or failure.

Example: A Function to Find Maximum Value

Function to Determine the Maximum of Two Integers

```
int max(int num1, int num2) {
    return (num1 > num2) ? num1 : num2;
}
```

Calling the Function

```
int a = 5, b = 10;
int maximum = max(a, b);
printf("Maximum: %d", maximum);
```

Embedded C Usage

Such a function could be used in an embedded system to determine the highest sensor value, control signal, or other measurement critical to the system's operation.

The Stack and Functions: How C Handles Calls

Understanding the Stack

Each function call in C is managed using a stack data structure that stores parameters, local variables, and return addresses.

Embedded C Consideration

Stack size is limited in embedded systems. Recursive functions or deep function calls can lead to stack overflow.

Recursion in Functions: Basics

What is Recursion?

Recursion occurs when a function calls itself to solve a problem by breaking it down into smaller, more manageable sub-problems.

Example: Recursive Function for Factorial

```
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Embedded C Note

Recursive functions should be used with caution in embedded systems due to limited stack space.

Example: Recursive Factorial Function

Full Recursive Factorial Program in C

```
#include <stdio.h>

int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

int main() {
    int num = 5;
    printf("Factorial of %d is %d", num, factorial(num));
    return 0;
}
```

Recursion vs. Iteration: Comparative Study

Comparing Recursion and Iteration

- Recursion can be more intuitive and easier to write for problems that naturally fit the recursive pattern.
- Iteration is generally more memory-efficient and can be faster because it does not incur the overhead of multiple function calls.

Embedded Systems Best Practice

Prefer iteration over recursion when working with resource-constrained embedded systems, unless recursion significantly simplifies the problem.

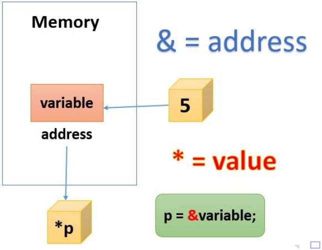
Introduction to Pointers

What is a Pointer?

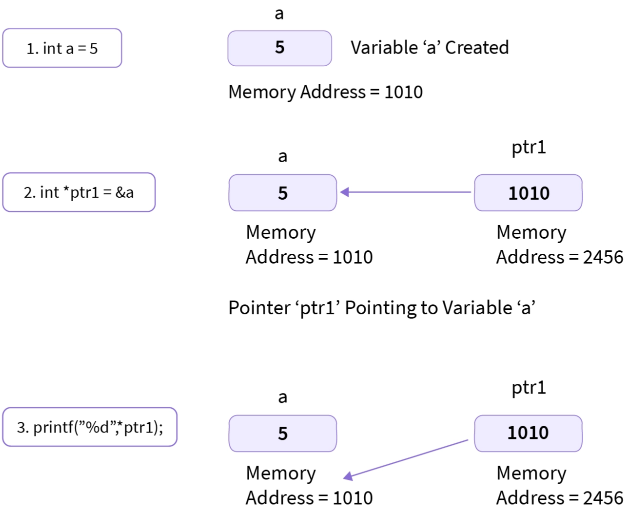
A pointer is a variable that stores the memory address of another variable. Pointers are a powerful feature in C that allow for dynamic memory management and efficient array handling.

Importance in Embedded Systems

Pointers are critical in embedded systems for interacting with hardware, managing memory, and optimizing performance.



Declaring and Using Pointers



When *ptr1 is called, it reads the memory address stored in ptr1 and goes to that memory address and reads the variable, i.e 5

Declaring and Using Pointers

Pointer Declaration

```
type *pointerName;
```

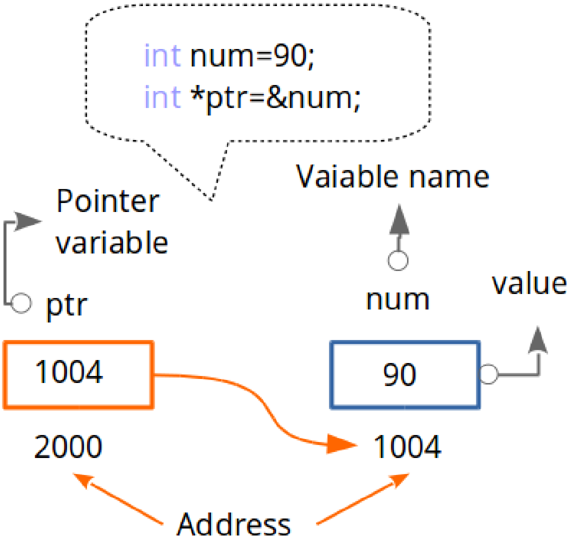
Pointer Usage

```
int var = 10;
int *ptr = &var;
```

Embedded C Example

```
char *bufferPtr;  // Pointer to a character buffer
```

Declaring and Using Pointers



Pointer Arithmetic

Pointer Operations

Pointer arithmetic allows pointers to be incremented or decremented, effectively moving through an array or block of memory.

Example: Navigating an Array

```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr;
for(int i = 0; i < 5; i++) {
    printf("%d ", *(ptr + i));
}
```

Pointers and Arrays

Relationship Between Pointers and Arrays

Arrays in C are closely related to pointers; the array name can be used as a pointer to the first element.

Example: Array Element Access

```
int array[3] = {1, 2, 3};
int *ptr = array;
printf("%d", *(ptr + 1));  // Outputs 2, the second element
```

Pointers and Strings

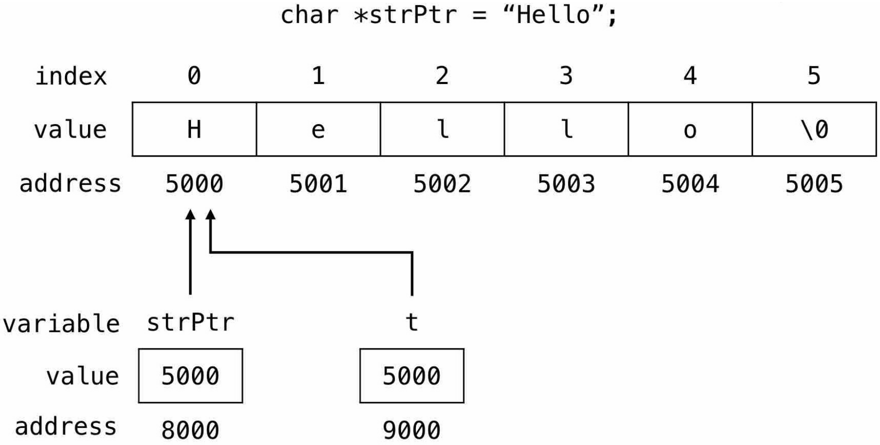
Using Pointers with Strings

Since strings are arrays of characters, pointers can be used to iterate and manipulate strings.

Example: String Traversal

```
char str[] = "Hello";
char *ptr = str;
while(*ptr != '\0') {
    putchar(*ptr++);
}
```

Pointers and Strings



Pointers in Functions: Pass-by-Reference

Pass-by-Reference Concept

Passing arguments by reference to a function allows the function to modify the original value.

Example: Modifying Variables

```
void increment(int *value) {
    (*value)++;
}
int main() {
    int num = 5;
    increment(&num);
    printf("%d", num); // Outputs 6
}
```

Embedded Systems Application

This technique is frequently used in embedded systems for updating hardware states or shared variables.

Example: Swapping Two Numbers Using Pointers

Swapping Function

```
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
int main() {
    int a = 10, b = 20;
    swap(&a, &b);
    printf("a: %d, b: %d", a, b); // Outputs a: 20, b: 10
}
```

Dynamic Memory Allocation in C

Heap Memory Allocation

Dynamic memory allocation involves managing memory at runtime using functions like ‘malloc‘, ‘calloc‘, ‘realloc‘, and ‘free‘.

Embedded Systems Consideration

Careful management of dynamic memory is crucial in embedded systems due to limited memory resources.

Structures: Custom Data Types

What is a Structure?

A structure in C is a user-defined data type that allows to combine data items of different kinds.

Use in Embedded Systems

Structures are extensively used in embedded systems for organizing complex data, like sensor readings or device configurations.

Defining and Declaring Structures

Structure Definition

```
struct MyStruct {
    int integer;
    char character;
};
```

Declaring a Structure Variable

```
struct MyStruct example;
example.integer = 5;
example.character = 'A';
```

Accessing Members of Structures

Accessing Structure Members

Members of a structure are accessed using the dot operator.

Example: Accessing and Modifying Members

```
struct MyStruct var;
var.integer = 10;
printf("Integer: %d", var.integer);
var.character = 'B';
```

Embedded Systems Note

Structures in embedded systems are often used to represent complex data structures like control registers or protocol frames.

Arrays of Structures

Using Arrays of Structures

Arrays of structures are useful for managing multiple sets of related data.

Example: Array of Structs

```
struct MyStruct array[2];
array[0].integer = 5;
array[0].character = 'X';
array[1].integer = 15;
array[1].character = 'Y';
```

Pointers to Structures

Working with Structure Pointers

Pointers can be used to access and manipulate structures, which is more efficient in terms of memory and performance.

Example: Accessing Structures Using Pointers

```
struct MyStruct obj;
struct MyStruct *ptr = &obj;
ptr->integer = 20;
printf("Integer through pointer: %d", ptr->integer);
```

Example: Sorting an Array of Structures

Implementing a Sorting Algorithm

Sorting an array of structures based on one of the member's values.

Example: Bubble Sort on Struct Array

```
// Assume struct MyStruct and an array of it are defined
// Implement a bubble sort algorithm to sort the array
// based on the integer member of the structures.
```

Unions in C: Basics

Introduction to Unions

A union is a special data type in C that allows storing different data types in the same memory location.

Use in Embedded Systems

Unions are useful in embedded systems for memory-efficient storage and for easy access to individual bytes of multi-byte data.

Defining and Using Unions

Union Definition

```
union MyUnion {
    int intVar;
    char charVar;
};
```

Using a Union

```
union MyUnion u;
u.intVar = 5;
printf("Integer: %d", u.intVar);
u.charVar = 'A';
printf("Character: %c", u.charVar);
```

Embedded Systems Application

Unions are used in embedded systems for accessing different types of data stored at the same memory location, such as sensor data.

Structures vs Unions: Memory Comparison

Memory Allocation

Structures allocate memory for each member separately, while unions share memory among all members, using the size of the largest member.

Example

A structure with an int and a char will have a size larger than the sum of both, whereas a union will have the size of the int, the larger member.

Considerations for Embedded Systems

Understanding how memory is allocated for structures and unions helps optimize memory usage in embedded systems.

Bit Fields in Structures for Memory Optimization

Using Bit Fields

Bit fields in structures allow for more memory-efficient storage by specifying the exact number of bits used for each member.

Example

```
struct {
    unsigned int lowVoltage: 1;
    unsigned int highTemperature: 1;
    unsigned int systemFailure: 1;
} statusFlags;
```

Embedded Systems Usage

This is particularly useful in embedded systems for packing multiple status flags or settings into a single byte.

Example: Using Unions for Type-Punning

Type-Punning with Unions

Type-punning involves accessing a data type as another type to interpret the data in different ways.

Example: Interpreting Int as Float

```
union {
    int intValue;
    float floatValue;
} pun;
pun.intValue = 0x40490fdb; // Representation of 3.14 in float
printf("Float value: %f", pun.floatValue);
```

Advanced String Manipulations

Complex String Operations

Discuss more complex string manipulations like substring extraction, pattern matching, and string tokenization.

Embedded Systems Context

In embedded systems, such operations might be used for parsing protocol messages, configuring settings, or displaying user interfaces.

String Parsing Techniques

Parsing Strings

String parsing involves breaking down a string into tokens or extracting specific information from it.

Common Techniques

- Using ‘strtok‘ for tokenizing strings.
- Extracting substrings using ‘substring‘ functions.
- Searching for patterns within strings.

Embedded Systems Application

Parsing sensor data formats or communication protocols are common tasks in embedded programming.

Implementing Custom String Functions

Creating Custom String Handlers

Developing custom string handling functions for specific needs that are not covered by standard library functions.

Example: Custom String Copy Function

```
void customStrCopy(char *dest, const char *src) {
    while (*src) {
        *dest++ = *src++;
    }
    *dest = '\0';
}
```

Embedded Systems Context

Custom string functions can be tailored for memory efficiency and specific data handling requirements in embedded systems.

Advanced Function Usage

Exploring Advanced Concepts

- Variable number of arguments with ‘stdarg.h‘.
- Using function pointers for callbacks and event handling.
- Inline functions for performance optimization.

Relevance in Embedded Systems

Such techniques can enhance flexibility and efficiency, important in resource-constrained embedded environments.

Inline Functions and Macros

Optimizing Performance

Inline functions and macros are used to reduce the overhead of function calls, particularly in small, frequently used functions.

Embedded Systems Optimization

Using inline functions and macros can lead to more efficient code, crucial for high-performance embedded systems.

Pointers to Functions: Basics

Function Pointers

A function pointer is a pointer that points to a function. This allows for dynamic function calls and passing functions as arguments to other functions.

Use Cases in Embedded Systems

Function pointers are extensively used for implementing callback mechanisms and interrupt service routines in embedded systems.

Example: Implementing a Callback Function

Callback Function Implementation

A callback function is passed to another function as an argument and is called within that function.

Example: Callback Function

```
void greet(void (*callback)(const char*)) {
    callback("Hello, World!");
}
void printMessage(const char* message) {
    printf("%s", message);
}
int main() {
    greet(printMessage);
    return 0;
}
```

Embedded Systems Context

Callback functions are often used in embedded systems for handling events like interrupts or sensor readings.

Memory Layout of a C Program

Understanding the Memory Layout

The memory layout of a C program is divided into segments like text, data, bss, heap, and stack.

Embedded Systems Consideration

Knowing the memory layout is crucial in embedded systems for optimizing memory usage and debugging memory-related issues.

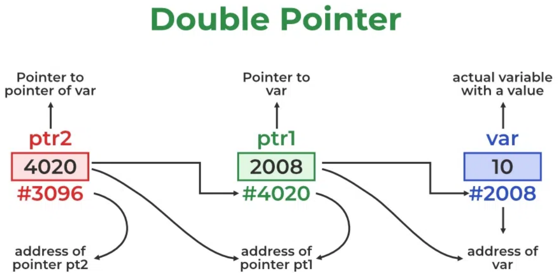
Understanding and Using Pointers to Pointers

Pointers to Pointers

A pointer to a pointer is a form of multiple indirection or a chain of pointers. Typically used for dynamic multi-dimensional arrays.

Application in Embedded Systems

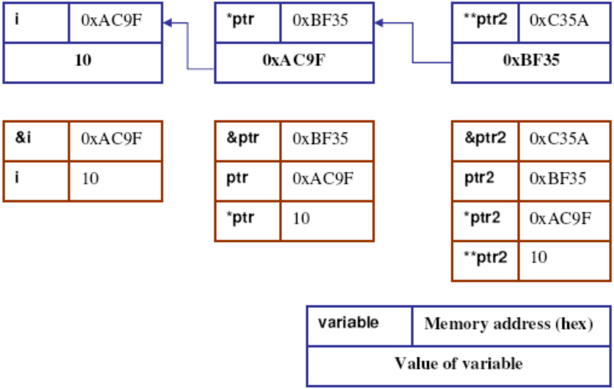
Pointers to pointers can be used in embedded systems for creating dynamic data structures like linked lists or buffer arrays.



Multi-Level Pointers and Their Uses

Advanced Pointer Concepts

Multi-level pointers, such as double or triple pointers, are used for complex data structures where levels of indirection add flexibility.



Structures and Pointers: Advanced Techniques

Combining Structures with Pointers

Structures can be dynamically allocated, manipulated, and passed to functions using pointers.

Embedded Systems Usage

This technique is essential for managing configuration data, device states, and protocol messages in embedded systems.

Nested Structures: Structures within Structures

Concept of Nested Structures

Nested structures are structures within structures, allowing for more complex data relationships and hierarchies.

Embedded Systems Application

They are useful for representing complex data in embedded systems, like a device with various sensors, each having its own set of attributes.

Example: Nested Structures for Complex Data

Defining and Using Nested Structures

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

struct Event {
    struct Date eventDate;
    char description[50];
};

struct Event myEvent = {{1, 1, 2022}, "New Year Celebration"};
```

Embedded Systems Context

This approach can be used for organizing configuration data, event logs, or complex state information.

Unions and Type-Punning: Advanced Concepts

Type-Punning with Unions

Type-punning using unions allows a single piece of memory to be interpreted in multiple ways, which is particularly useful in low-level programming.

Embedded Systems Implication

Useful for protocol handling, where the same bytes might be interpreted differently based on the context.

Pointers and Dynamic Memory: Advanced Uses

Dynamic Memory in C

Pointers are integral to dynamic memory management in C, providing flexibility and control over memory allocation.

Considerations for Embedded Systems

While powerful, dynamic memory allocation must be used judiciously in embedded systems due to limited memory resources and the need for deterministic behavior.

Memory Leaks and Pointer Safety

Handling Memory Leaks

Memory leaks occur when dynamically allocated memory is not freed properly. Proper management is crucial to prevent memory waste and potential crashes.

Safe Pointer Practices

Use of pointers must be done with care to ensure memory safety, including proper initialization, bounds checking, and freeing allocated memory.

Example: Building a Linked List

Linked List in C

A linked list is a dynamic data structure that can grow and shrink at runtime. It consists of nodes that contain data and a pointer to the next node.

Defining a Node

```
struct Node {
    int data;
    struct Node *next;
};
```

Embedded Systems Context

Linked lists are useful for managing dynamic collections of data like event logs or task queues in embedded systems.

Function Pointers and Event-Driven Programming

Function Pointers for Flexibility

Function pointers can be used to implement event-driven programming by associating functions with specific events or interrupts.

Application in Embedded Systems

This approach is widely used in embedded systems for handling hardware interrupts, timers, and other event-driven mechanisms.

Pointers and Memory: Best Practices

Ensuring Safe Pointer Usage

- Always initialize pointers.
- Avoid pointer arithmetic errors.
- Be cautious with pointer casting.
- Ensure proper memory allocation and deallocation.

Considerations for Embedded Development

Pointer-related errors can be particularly critical in embedded systems where they can lead to system crashes or unpredictable behavior.

Structures, Unions, and Endianness

Understanding Endianness

Endianness refers to the order of bytes in multi-byte data types. Structures and unions must be used carefully to account for endianness in data communication.

Embedded Systems Implications

Correct handling of endianness is crucial in embedded systems, especially in network communications and data storage.

Example: Endianness Conversion

Implementing Endianness Conversion

Functions to convert between big-endian and little-endian representations are important in systems where data interchange formats vary.

Example Function

```
uint16_t convertEndian(uint16_t value) {
    return (value >> 8) | (value << 8);
}
```

Debugging Tips for Pointer-Related Issues

Identifying and Resolving Pointer Issues

- Use debugging tools to track pointer values and memory addresses.
- Check for null pointers before dereferencing.
- Be cautious of memory leaks and dangling pointers.
- Use memory profilers to identify and fix memory-related issues.

Embedded Systems Context

Debugging pointer issues in embedded systems can be challenging due to limited debugging interfaces and real-time constraints.

Memory Constraints and Data Alignment

Handling Memory in Embedded Systems

- Understanding the limitations of available memory.
- The importance of data alignment for efficient access and storage.
- Techniques for memory optimization in constrained environments.

Example: Custom Memory Allocator

Developing a Custom Memory Allocator

Designing and implementing a memory allocation strategy tailored for specific requirements of an embedded system.

Example Code Snippet

```
// Pseudocode or C code demonstrating a simple
// custom memory allocator, managing a fixed-size buffer
// for dynamic allocation within an embedded system.
```

Pointer Challenge 1

Challenge

Given an array of integers, write a function to reverse the array using pointers.

Solution to Pointer Challenge 1

Solution

```
void reverseArray(int *arr, int size) {
    int *start = arr;
    int *end = arr + size - 1;
    while (start < end) {
        int temp = *start;
        *start++ = *end;
        *end-- = temp;
    }
}
```

Pointer Challenge 2

Challenge

Write a C program to find the length of a string using a pointer.

Solution to Pointer Challenge 2

Solution

```c
int stringLength(char *str) {
    char *ptr = str;
    int len = 0;
    while (*ptr != '\0') {
        len++;
        ptr++;
    }
    return len;
}
```

Pointer Challenge 3

Challenge

Create a function using pointers to swap the values of two integers.

Solution to Pointer Challenge 3

Solution

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Pointer Challenge 4

Challenge

Given a pointer to the start of an integer array, write a function to compute the sum of its elements.

Solution to Pointer Challenge 4

Solution

```c
int arraySum(int *arr, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += *(arr + i);
    }
    return sum;
}
```

Pointer Challenge 5

Challenge

Write a C function to concatenate two strings using pointers.

Solution to Pointer Challenge 5

Solution

```
void concatenate(char *dest, const char *src) {
    while (*dest) dest++;
    while (*src) *dest++ = *src++;
    *dest = '\0';
}
```

Real-Time Scenario 1: Sensor Data Processing

Scenario Description

Develop a function in Embedded C to process data from multiple sensors. Each sensor's data is stored in an array. The function should calculate the average value of each sensor's data.

Embedded C Application

Sensor data processing is a common task in embedded systems for applications like environmental monitoring or system diagnostics.

Solution to Real-Time Scenario 1

Embedded C Code Snippet

```
float calculateAverage(int *data, int size) {
    int sum = 0;
    for(int i = 0; i < size; i++) {
        sum += data[i];
    }
    return (float)sum / size;
}
```

Explanation

This function iterates over an array of sensor readings, calculates the total sum, and then returns the average.

Real-Time Scenario 2: Buffer Management

Scenario Description

Implement a buffer management system in Embedded C to store and retrieve messages from a communication interface, ensuring data integrity and efficient memory usage.

Embedded C Significance

Effective buffer management is crucial in embedded systems for handling data communication and preventing buffer overflows or data loss.

Solution to Real-Time Scenario 2

Embedded C Code Snippet

```
#define BUFFER_SIZE 100
char buffer[BUFFER_SIZE];
int head = 0, tail = 0;

void addToBuffer(char data) {
    buffer[tail] = data;
    tail = (tail + 1) % BUFFER_SIZE;
}

char readFromBuffer() {
    char data = buffer[head];
    head = (head + 1) % BUFFER_SIZE;
    return data;
}
```

Explanation

A circular buffer implementation to efficiently manage data in a fixed-size buffer.

Real-Time Scenario 3: Device Control Protocol

Scenario Description

Create a protocol in Embedded C to control various devices connected to a microcontroller, using function pointers for modularity and ease of maintenance.

Embedded C Context

Device control protocols are essential in embedded systems for managing multiple devices and their operations.

Solution to Real-Time Scenario 3

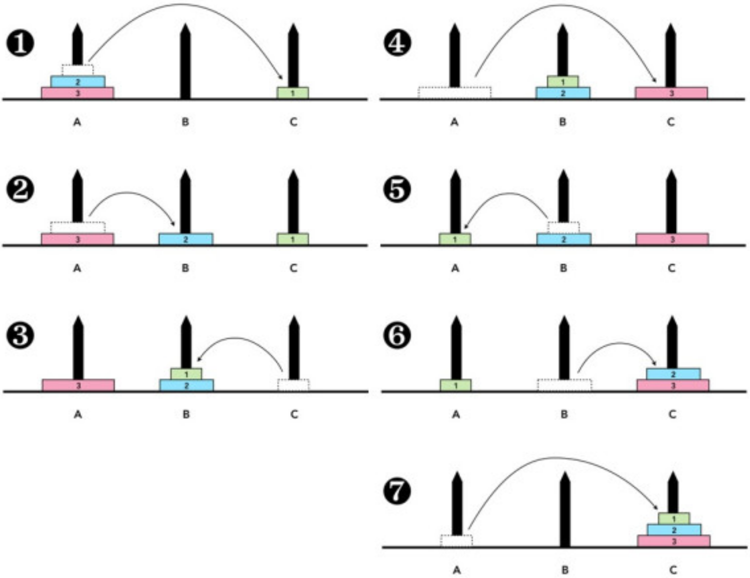
Embedded C Code for Device Control Protocol

```
void controlLED(int command);
void controlMotor(int command);
void controlSensor(int command);

void (*deviceControl[])(int) = {controlLED, controlMotor, cont

void controlDevice(int device, int command) {
    (*deviceControl[device])(command);
}
// Example usage: controlDevice(0, ON); // Turn on the LED
```

Explanation

This implementation uses an array of function pointers for different device control functions, allowing for flexible and modular device management.

Tower of Hanoi Problem



Tower of Hanoi Problem

- Objective: Move all disks from one peg to another, with only one disk moved at a time, and a larger disk cannot be placed on top of a smaller disk.
- Uses recursion to solve the problem elegantly.
- Implementation in C demonstrates arrays for pegs, recursive function calls, and visual representation of the pegs' state.

C Program Highlights:

- `printPegs` function to display the pegs.
- `moveDisk` function to move a disk from one peg to another.
- Recursive `towerOfHanoi` function to solve the problem.
- 2D array `pegs` to represent the state of each peg.

Example Usage:

- Initial setup with `n` disks on the first peg.
- Recursive calls to move disks between pegs.
- Visual output after each move.

References I

Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice Hall, 2nd Edition, 1988.
The definitive guide to C programming by its original creators.

Stephen Prata.
C Primer Plus.
Pearson Education, 6th Edition, 2013.
Comprehensive guide to C programming, covering basic to advanced topics.

References II

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
Introduction to Algorithms.
MIT Press, 3rd Edition, 2009.
Provides insights into algorithmic thinking relevant to programming challenges.

Michael Barr.
Programming Embedded Systems in C and C++.
O'Reilly Media, 1999.
A book focusing on embedded systems programming.

Michael J. Pont.
Patterns for Time-Triggered Embedded Systems.
Addison-Wesley, 2001.
Building reliable applications with the 8051 family of microcontrollers.

References III

- GeeksforGeeks - C Programming Language.
  https://www.geeksforgeeks.org/c-programming-language/
  A comprehensive online resource for learning C with examples and tutorials.
- Learn C and C++ Programming - Cprogramming.com.
  https://www.cprogramming.com/
  An online portal offering tutorials and explanations on C and C++ programming.