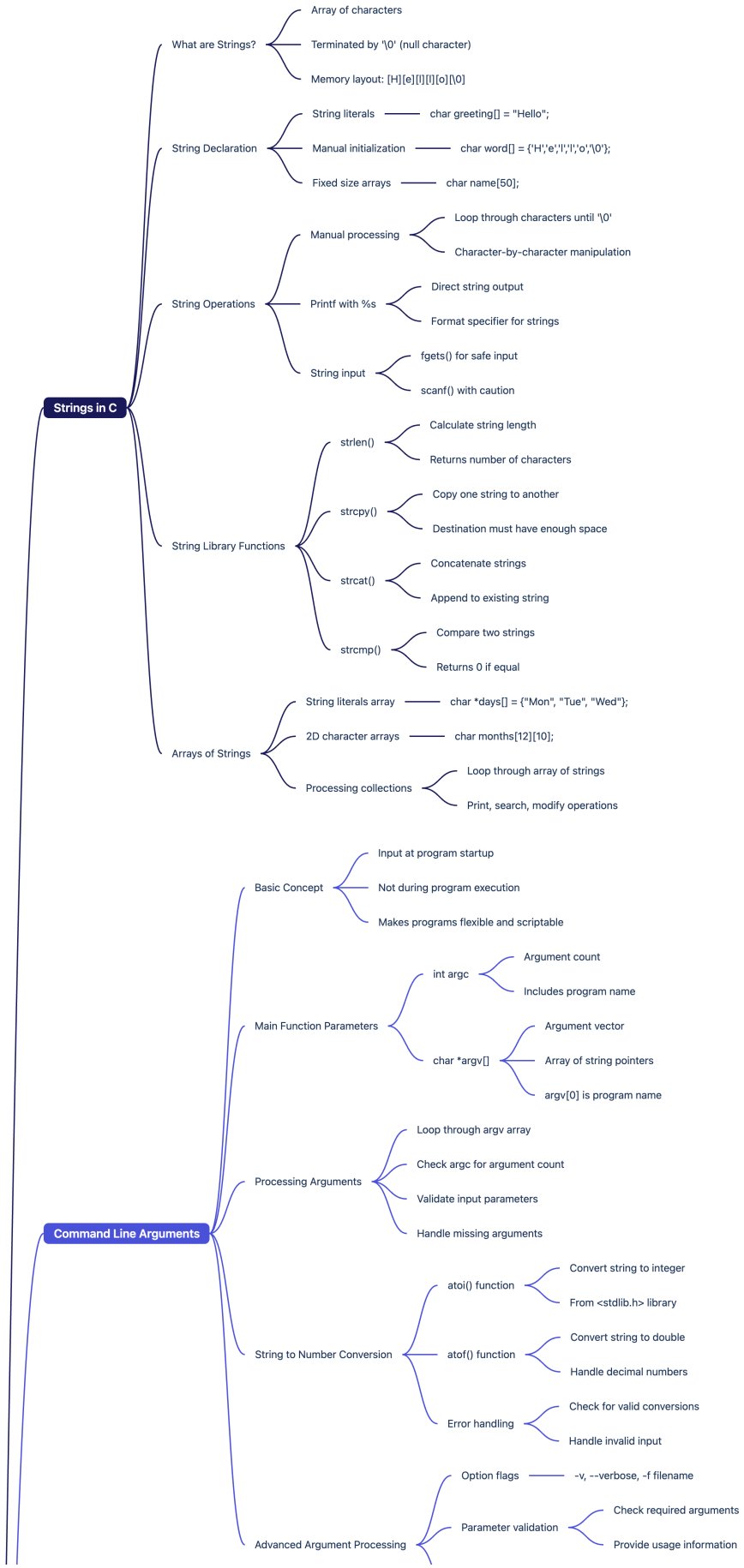
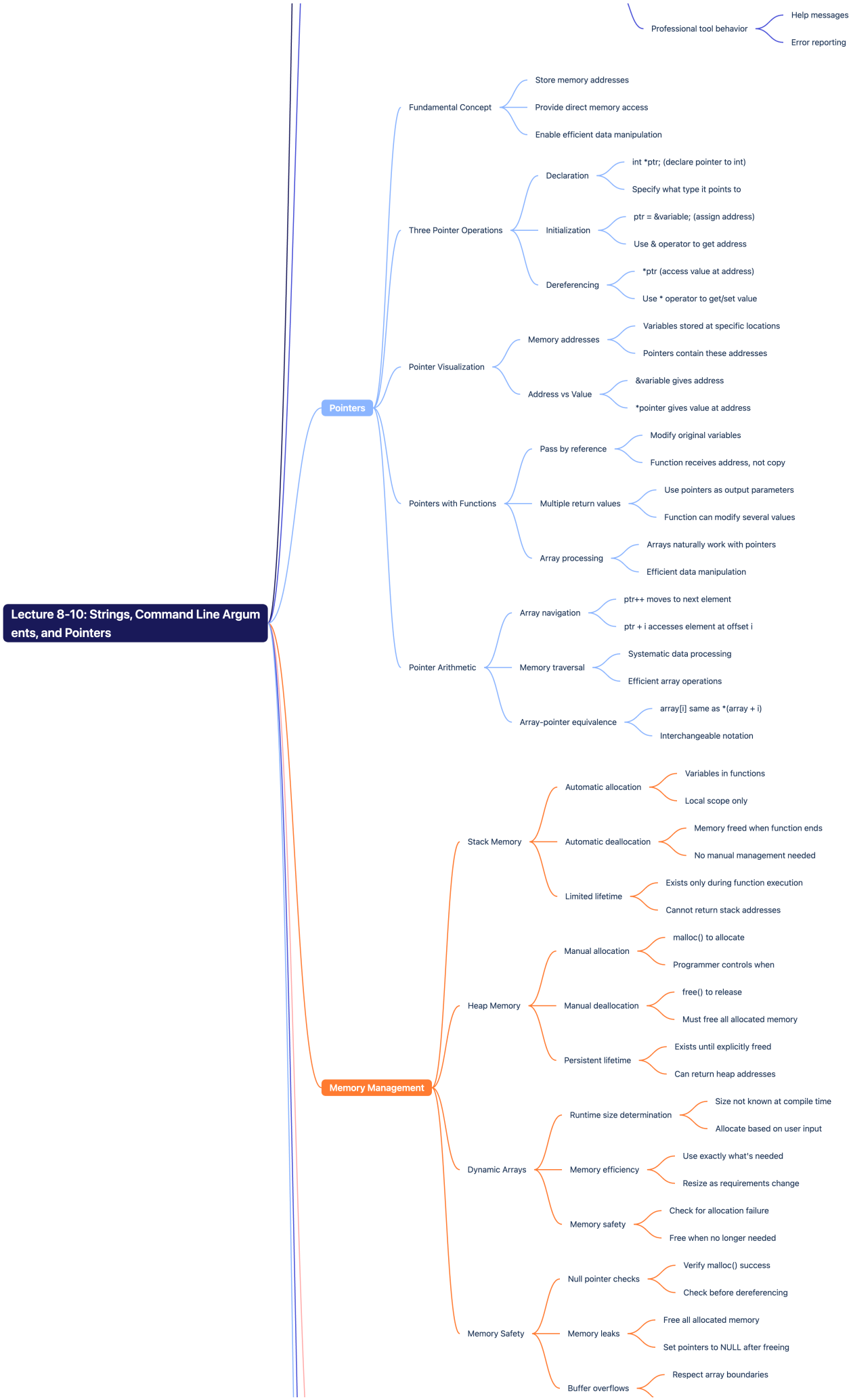
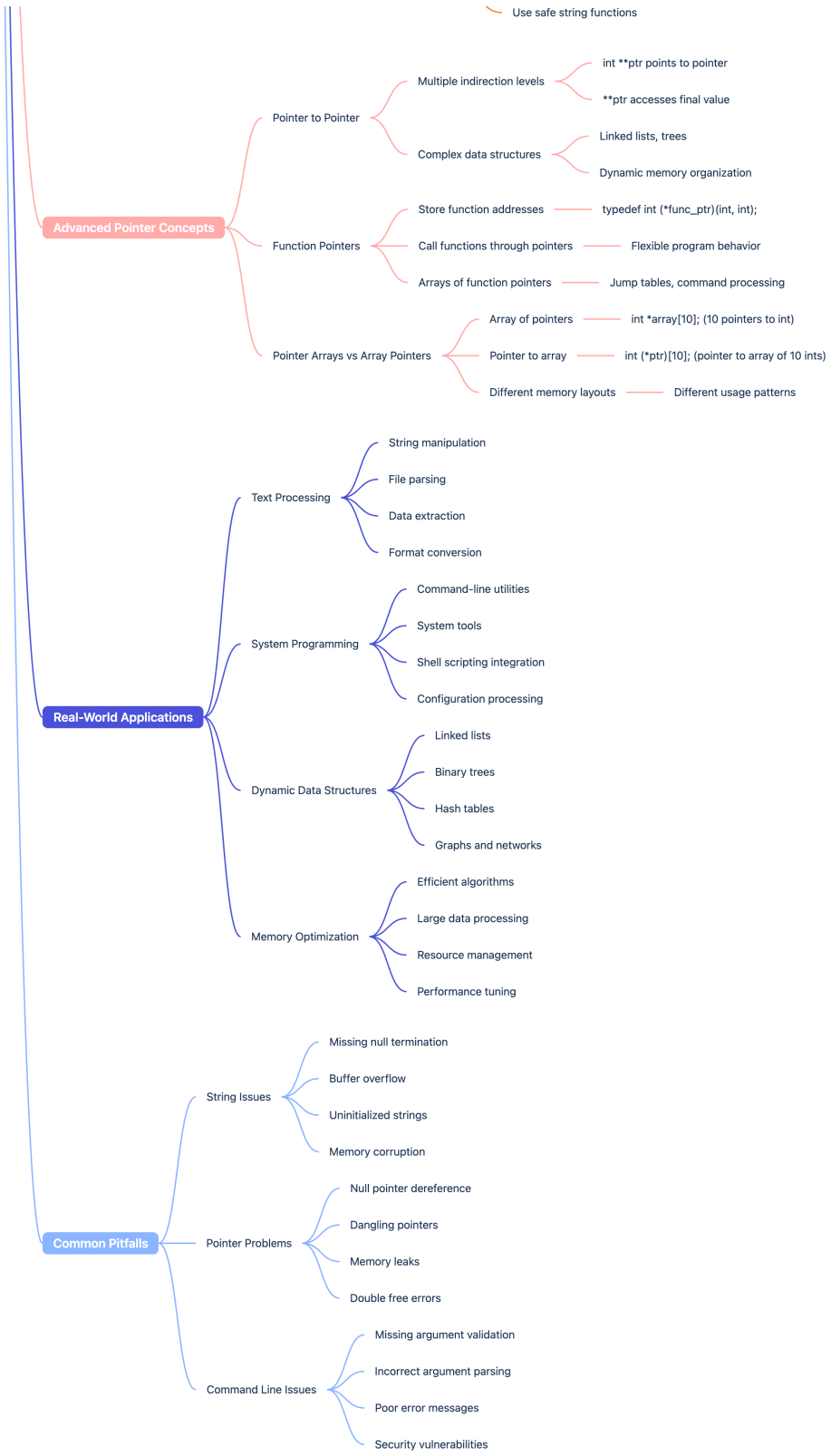
Lecture 7-9: Strings, Command Line Arguments, and Pointers - The Deep Dive




Lecture 8-10: Strings, Command Line Arguments, and Pointers
Pointers
Fundamental Concept
Store memory addresses
Provide direct memory access
Enable efficient data manipulation
Three Pointer Operations
Declaration
int *ptr; (declare pointer to int)
Specify what type it points to
Initialization
ptr = &variable; (assign address)
Use & operator to get address
Dereferencing
*ptr (access value at address)
Use * operator to get/set value
Pointer Visualization
Memory addresses
Variables stored at specific locations
Pointers contain these addresses
Address vs Value
&variable gives address
*pointer gives value at address
Pointers with Functions
Pass by reference
Modify original variables
Function receives address, not copy
Multiple return values
Use pointers as output parameters
Function can modify several values
Array processing
Arrays naturally work with pointers
Efficient data manipulation
Pointer Arithmetic
Array navigation
ptr++ moves to next element
ptr + i accesses element at offset i
Memory traversal
Systematic data processing
Efficient array operations
Array-pointer equivalence
array[i] same as *(array + i)
Interchangeable notation
Professional tool behavior
Help messages
Error reporting
Memory Management
Stack Memory
Automatic allocation
Variables in functions
Local scope only
Automatic deallocation
Memory freed when function ends
No manual management needed
Limited lifetime
Exists only during function execution
Cannot return stack addresses
Heap Memory
Manual allocation
malloc() to allocate
Programmer controls when
Manual deallocation
free() to release
Must free all allocated memory
Persistent lifetime
Exists until explicitly freed
Can return heap addresses
Dynamic Arrays
Runtime size determination
Size not known at compile time
Allocate based on user input
Memory efficiency
Use exactly what's needed
Resize as requirements change
Memory safety
Check for allocation failure
Free when no longer needed
Memory Safety
Null pointer checks
Verify malloc() success
Check before dereferencing
Memory leaks
Free all allocated memory
Set pointers to NULL after freeing
Buffer overflows
Respect array boundaries



Why These Topics Matter

These advanced concepts unlock the true power of C programming and bridge the gap to professional software development. Strings let programs work with text data naturally, command line arguments make programs flexible and professional, and pointers provide the foundation for advanced programming techniques. Together, they form the gateway to systems programming, dynamic memory management, and understanding how computers really work under the hood.

Discussion: What's the difference between a simple program that only works one way and a professional tool that can be customized? Think about the apps and tools you use daily.

1. Strings: The Computer's Language Skills

Understanding Text in the Computer's World

Think of strings as the computer's way of understanding and working with human language. Just like how we write sentences as sequences of letters, the computer stores text as sequences of characters. But there's a secret - the computer needs a special "end of sentence" marker (the null terminator '\0') to know where each piece of text finishes. It's like how we use periods to end sentences, except the computer uses an invisible marker that says "the text stops here."

Discussion: How do you think computers store and process all the text in your favourite apps? (Messages, social media, documents, etc.)

String Fundamentals: The Text Storage System

```
// String is like a box of letters with a special "end" marker
char greeting[] = "Hello";  // Convenient way - C automatically adds the
'\0'
char word[] = {'H','e','l','l','o','\0'};  // Manual way - we add the '\0'
ourselves

// Memory layout - like looking inside the computer's filing system:
// [0] [1] [2] [3] [4] [5]
//  H   e   l   l   o  \0
// Each box holds one character, and '\0' marks the end
```

The null terminator '\0' is like the computer's full stop. Without it, the computer would keep reading memory until it randomly found a zero, potentially reading garbage data. It's like reading a book where someone forgot to put periods - you wouldn't know where sentences end.

Exercise: What would happen if we forgot the '\0' in the manual initialization above?

String Operations: Working with Text

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

// Manual string printing - like reading each letter one by one
void print_string_manual(char str[]) {
    int i = 0;  // Start at the first character
    while (str[i] != '\0') {  // Keep going until we hit the end marker
        printf("%c", str[i]);  // Print one character at a time
        i++;  // Move to the next character
    }
    printf("\n");  // Add a new line when we're done
}
```

```c
// Using printf with %s - let printf do all the work for us
void print_string_easy(char str[]) {
    printf("%s\n", str);  // printf knows to stop at '\0'
}

// String input - like giving the computer a form to fill out
char name[50];  // Create space for up to 49 characters plus '\0'
printf("Enter your name: ");
fgets(name, 50, stdin);  // Safer than scanf - won't overflow our buffer
```

Challenge: Why is `fgets` safer than `scanf` for reading strings?

String Functions: The Text Toolkit

```c
#include <string.h>

int main(void) {
    char str1[20] = "Hello";  // First string - has room for 19 chars +
'\0'
    char str2[20] = "World";  // Second string - same size for safety

    // String length - like counting letters in a word
    int len = strlen(str1);  // Counts characters until it hits '\0'
    printf("Length: %d\n", len);  // Will print 5 (doesn't count '\0')

    // String copy - like photocopying text from one page to another
    strcpy(str2, str1);  // str2 now contains "Hello" (overwrites "World")

    // String concatenation - like gluing two words together
    strcat(str1, " World");  // str1 now "Hello World" (adds to the end)

    // String comparison - like checking if two words are spelled the same
    if (strcmp(str1, str2) == 0) {  // Returns 0 if strings are identical
        printf("Strings are equal\n");
    } else {
        printf("Strings are different\n");  // This will print
    }

    return 0;
}
```

Exercise: What will `str1` and `str2` contain at the end of this program?

String Functions Table

Function	What It Does	Real-World Analogy	Example
`strlen(str)`	Counts characters (excluding '\0')	Counting letters in a word	`strlen("Hello")` returns 5

Function	What It Does	Real-World Analogy	Example
`strcpy(dest, src)`	Copies one string to another	Photocopying text	`strcpy(name, "Alice")`
`strcat(dest, src)`	Adds one string to the end of another	Gluing two words together	`strcat("Hello", " World")`
`strcmp(str1, str2)`	Compares two strings	Checking if words are identical	Returns 0 if equal

Question: Why doesn't `strlen` count the null terminator in its result?

Arrays of Strings: The Text Collection

```c
// Array of string literals - like a list of ready-made labels
char *days[] = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};

// 2D character array - like a filing cabinet with fixed-size folders
char months[12][10] = {"January", "February", "March", "April"};

// Processing array of strings - like reading through a list
void print_days(char *days[], int size) {
    for (int i = 0; i < size; i++) {  // Go through each string in the
array
        printf("Day %d: %s\n", i + 1, days[i]);  // Print each day with a
number
    }
}

int main(void) {
    print_days(days, 5);  // Show all 5 weekdays
    return 0;
}
```

Think of arrays of strings like having a bookshelf where each book represents one piece of text. You can access any book (string) by its position on the shelf. The `char *days[]` approach is like having adjustable shelves that fit exactly the size of each book, while `char months[12][10]` is like having fixed-size slots that can hold books up to a certain size.

Challenge: What's the advantage of each approach? When would you use `char *[]` vs `char[][]`?

2. Command Line Arguments: Making Programs Professional

The Professional Program Interface

Command line arguments are like giving your program a briefing before it starts work. Instead of asking the user questions while running, you tell the program everything it needs to know right from the start. It's like the difference between a waiter who takes your order at the table versus a food truck where you tell them exactly what you want when you walk up to the window.

Discussion: What command line programs do you use? (ls, cd, cp, mv, git, etc.) How do they accept different options and parameters?

Basic Command Line Arguments: The Program's Starting Instructions

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

// argc = argument count (how many pieces of information were given)
// argv = argument vector (an array containing each piece of information)
int main(int argc, char *argv[]) {
    printf("Program name: %s\n", argv[0]);  // argv[0] is always the
program name
    printf("Number of arguments: %d\n", argc);  // argc includes the
program name

    // Loop through all arguments (skip argv[0] since that's just the
program name)
    for (int i = 1; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);  // Print each argument
    }

    return 0;
}
```

Think of `argc` as the count of how many items are in your shopping bag, and `argv` as the actual shopping bag containing all the items. The first item (`argv[0]`) is always the receipt showing what store you shopped at (the program name).

Exercise: If you run this program with `./myprogram hello world 123`, what will `argc` be and what will each `argv` contain?

Converting String Arguments to Numbers: The Data Translator

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

int main(int argc, char *argv[]) {
    // Check if the user gave us the right number of arguments
    if (argc != 3) {  // We expect: program name, number1, number2 (total
= 3)
        printf("Usage: %s <number1> <number2>\n", argv[0]);
        return 1;  // Exit with error code
    }

    // Convert strings to integers - like translating text to numbers
    int num1 = atoi(argv[1]);  // atoi = "ASCII to Integer"
    int num2 = atoi(argv[2]);  // Converts "123" string to 123 number

    int sum = num1 + num2;  // Now we can do math with them
```

```
    printf("%d + %d = %d\n", num1, num2, sum);

    return 0;  // Success!
}
```

The `atoi` function is like having a translator who can read numbers written as text and convert them to actual numbers the computer can do math with. When someone types "42" on the command line, it comes in as the characters '4' and '2', not the number 42. The `atoi` function does the conversion.

Question: What happens if someone types "hello" instead of a number? How could we make our program more robust?

Advanced Command Line Processing: The Professional Interface

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

int main(int argc, char *argv[]) {
    int verbose = 0;        // Flag to track if verbose mode is enabled
    char *filename = NULL;  // Pointer to store filename (if provided)

    // Process command line options - like reading a list of instructions
    for (int i = 1; i < argc; i++) {  // Start at 1 to skip program name
        if (strcmp(argv[i], "-v") == 0 || strcmp(argv[i], "--verbose") ==
0) {
            verbose = 1;  // Turn on verbose mode
        } else if (strcmp(argv[i], "-f") == 0 && i + 1 < argc) {
            filename = argv[i + 1];  // Next argument is the filename
            i++;  // Skip next argument since we just used it
        } else {
            printf("Unknown option: %s\n", argv[i]);
            return 1;  // Exit with error for unknown options
        }
    }

    // Use the settings we found
    if (verbose) {
        printf("Verbose mode enabled\n");
    }

    if (filename) {
        printf("Processing file: %s\n", filename);
    }

    return 0;
}
```

This is like creating a smart assistant that can understand different types of requests. The program can handle flags (like `-v` for verbose) and options with values (like `-f filename`). It's similar to how

professional command-line tools work - they're flexible and can be customized for different situations.

Challenge: How would you add a `-h` or `--help` option that shows usage information?

3. Pointers: The Computer's Address System

Understanding Memory Addresses

Think of pointers as the computer's addressing system, like postal addresses for houses on a street. Every piece of data in your computer's memory has a unique address, just like every house has a unique street address. A pointer is like writing down someone's address on a piece of paper - it tells you where to find something, but it's not the thing itself. When you want to visit the house, you follow the address to get there.

Discussion: How is a pointer similar to a bookmark in a book or a shortcut on your desktop?

Pointer Basics: The Address Book System

```c
#include <stdio.h>

int main(void) {
    int number = 42;        // A variable - like a house with the number 42 inside
    int *number_ptr;        // A pointer - like a piece of paper to write an address

    number_ptr = &number;   // Write down the address of 'number' on our paper

    printf("Value of number: %d\n", number);           // What's inside the house
    printf("Address of number: %p\n", &number);        // The house's address
    printf("Value of number_ptr: %p\n", number_ptr);   // Address written on our paper
    printf("Value pointed to by number_ptr: %d\n", *number_ptr);  // Follow the address to see what's inside

    // Modify through pointer - like sending mail to change what's in the house
    *number_ptr = 100;      // Go to the address and change the contents
    printf("New value of number: %d\n", number);  // The original variable changed!

    return 0;
}
```

Key Pointer Operations:

- `&variable` - "Give me the address of this variable" (like asking for someone's home address)
- `*pointer` - "Go to this address and get/set what's there" (like visiting the house at that address)

- `pointer = &variable` - "Write down this address" (like copying an address to your address book)

Exercise: If `number` lives at address 1000, what will `number_ptr` contain? What will `*number_ptr` give us?

Pointers with Functions: The Powerful Partnership

```c
// Function that modifies values through pointers - like a trading service
void swap(int *a, int *b) {
    int temp = *a;  // Remember what's at address 'a'
    *a = *b;        // Put what's at address 'b' into address 'a'
    *b = temp;      // Put the remembered value into address 'b'
}

// Function that returns multiple values through pointers - like a survey
that fills out a report
void calculate_stats(int array[], int size, int *min, int *max, double
*average) {
    *min = array[0];  // Start by assuming first number is smallest
    *max = array[0];  // Start by assuming first number is largest
    int sum = 0;      // Keep track of total for average

    for (int i = 0; i < size; i++) {  // Check each number in the array
        if (array[i] < *min) *min = array[i];  // Found a smaller number
        if (array[i] > *max) *max = array[i];  // Found a larger number
        sum += array[i];  // Add to our running total
    }

    *average = (double)sum / size;  // Calculate and store the average
}

int main(void) {
    int x = 10, y = 20;
    printf("Before swap: x=%d, y=%d\n", x, y);
    swap(&x, &y);  // Give the function the addresses of x and y
    printf("After swap: x=%d, y=%d\n", x, y);  // x and y have been
swapped!

    int numbers[] = {5, 2, 8, 1, 9};
    int min, max;     // Variables to receive the results
    double avg;

    calculate_stats(numbers, 5, &min, &max, &avg);  // Give addresses for
results
    printf("Min: %d, Max: %d, Average: %.2lf\n", min, max, avg);

    return 0;
}
```

Pointers with functions solve the "one return value" limitation. It's like hiring a contractor who can do multiple jobs at your house - instead of calling them once for each task, you give them your address and a

list of what needs to be done, and they complete everything in one visit.

Question: Why can't we just return multiple values from a function normally? How do pointers solve this problem?

4. Memory Management: The Computer's Real Estate

Stack vs Heap Memory: Two Different Neighborhoods

Think of computer memory like a city with two different neighborhoods. The **stack** is like a temporary parking garage - you get a space automatically when you arrive, and it's freed up automatically when you leave. The **heap** is like renting an apartment - you have to specifically ask for space, use it as long as you need, and remember to give it back when you're done.

Discussion: What happens in real life when someone forgets to return borrowed items? How might this apply to computer memory?

Stack vs Heap Memory: The Two Memory Systems

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

// Stack allocation (automatic) - like temporary parking
void stack_example() {
    int local_array[100];  // Space automatically reserved on the stack
    // When this function ends, the space is automatically returned
    // Like leaving the parking garage - your spot becomes available again
}

// Heap allocation (dynamic) - like renting an apartment
int* create_array(int size) {
    // Ask the system for space to store 'size' integers
    int *array = malloc(size * sizeof(int));  // malloc = "memory
allocate"

    if (array == NULL) {  // Check if we got the space we asked for
        printf("Memory allocation failed\n");  // Like "apartment not
available"
        return NULL;
    }

    // Initialize array - like furnishing your new apartment
    for (int i = 0; i < size; i++) {
        array[i] = i * i;  // Fill with square numbers: 0, 1, 4, 9, 16...
    }

    return array;  // Give back the address of our rented space
}

int main(void) {
    int size = 10;
    int *dynamic_array = create_array(size);  // Rent some memory space
```

```
    if (dynamic_array != NULL) {  // Make sure we got the space
        for (int i = 0; i < size; i++) {  // Use our rented space
            printf("%d ", dynamic_array[i]);
        }
        printf("\n");

        free(dynamic_array);  // Return the rented space - very important!
        dynamic_array = NULL; // Forget the address (good practice)
    }

    return 0;
}
```

Memory Management Rules:

- **Stack**: Automatic cleanup, limited size, fast access
- **Heap**: Manual cleanup required, large size available, slightly slower access
- **Golden Rule**: Every `malloc()` must have a matching `free()`

Exercise: What happens if we forget to call `free()`? What happens if we call `free()` twice?

Dynamic String Handling: Custom Text Creation

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

// Function that creates a custom greeting - like a personalized card
maker
char* create_greeting(char *name) {
    // Rent space for 50 characters (enough for most greetings)
    char *greeting = malloc(50 * sizeof(char));

    if (greeting == NULL) {  // Check if we got the space
        return NULL;  // Couldn't create the greeting
    }

    // Build the greeting step by step
    strcpy(greeting, "Hello, ");  // Start with "Hello, "
    strcat(greeting, name);       // Add the person's name
    strcat(greeting, "!");        // Add an exclamation mark

    return greeting;  // Give back the address of our custom message
}

int main(void) {
    char name[20];
    printf("Enter your name: ");
    scanf("%s", name);  // Get the user's name
```

```
    char *message = create_greeting(name);  // Create a personalized
greeting
    if (message != NULL) {  // Make sure the greeting was created
successfully
        printf("%s\n", message);  // Show the greeting
        free(message);  // Don't forget to return the rented memory!
    }

    return 0;
}
```

This demonstrates dynamic string creation - like having a custom greeting card service that creates personalized messages. The function rents memory space, builds a custom string, and returns it. The caller is responsible for freeing the memory when done.

Challenge: How could you modify this function to handle very long names safely?

5. Practical Applications: Real-World Text and Memory Management

String Processing Toolkit: The Text Swiss Army Knife

You're building a text processing toolkit - like having a Swiss Army knife for working with text. This toolkit can count words, reverse text, change capitalization, check for palindromes, and remove spaces. It's the kind of utility that could be part of a word processor or text editor. **Key Functions Demonstrated:**

```
// Count words - like a word processor's word count feature
int count_words(char str[]) {
    int count = 0;
    int in_word = 0;  // Track whether we're currently inside a word

    for (int i = 0; str[i] != '\0'; i++) {  // Check each character
        if (str[i] != ' ' && str[i] != '\t' && str[i] != '\n') {  // Not
whitespace
            if (!in_word) {  // Starting a new word
                count++;
                in_word = 1;  // Now we're inside a word
            }
        } else {
            in_word = 0;  // Hit whitespace, no longer in a word
        }
    }
    return count;
}

// Reverse string - like reading text backwards
void reverse_string(char str[]) {
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {  // Only go halfway
        char temp = str[i];                    // Remember first character
        str[i] = str[len - 1 - i];            // Put last character at
start
```

```
        str[len - 1 - i] = temp;              // Put first character at
end
    }
}

// Check palindrome - like testing if text reads the same forwards and
backwards
int is_palindrome(char str[]) {
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {  // Compare first half with second
half
        if (tolower(str[i]) != tolower(str[len - 1 - i])) {  // Characters
don't match
            return 0;  // Not a palindrome
        }
    }
    return 1;  // All characters matched - it's a palindrome!
}
```

Question: How would you implement a function that converts text to "leet speak" (replacing 'e' with '3', 'a' with '@', etc.)?

Command Line Calculator: The Professional Math Tool

You're creating a command-line calculator that works like professional developer tools. Instead of asking users for input while running, it gets everything it needs from the command line arguments. This makes it scriptable and perfect for automation. **Key Concepts Demonstrated:**

```
// Main function with command line arguments
int main(int argc, char *argv[]) {
    if (argc != 4) {  // Need exactly 4: program name, num1, operation,
num2
        print_usage(argv[0]);  // Show how to use the program
        return 1;  // Exit with error code
    }

    double num1 = atof(argv[1]);  // Convert first number from string to
double
    char *operation = argv[2];    // Operation is already a string
    double num2 = atof(argv[3]);  // Convert second number from string to
double

    double result = perform_operation(operation, num1, num2);  // Do the
math
    printf("%.2lf %s %.2lf = %.2lf\n", num1, operation, num2, result);

    return 0;
}

// Function that handles different operations - like a smart calculator
double perform_operation(char *operation, double a, double b) {
```

```
    if (strcmp(operation, "+") == 0) {  // Addition
        return a + b;
    } else if (strcmp(operation, "*") == 0) {  // Multiplication
        return a * b;
    } else if (strcmp(operation, "/") == 0) {  // Division
        if (b == 0) {  // Can't divide by zero!
            printf("Error: Division by zero\n");
            return 0;  // Return error value
        }
        return a / b;
    } else {
        printf("Error: Unknown operation '%s'\n", operation);
        return 0;  // Return error value
    }
}
```

Usage Example: `./calculator 15.5 + 24.3` would output `15.50 + 24.30 = 39.80`

Exercise: How would you add support for parentheses or multiple operations?

Dynamic Array Manager: The Flexible Data Container

You're building a dynamic array manager - like having a smart storage system that can grow and shrink as needed. This demonstrates the power of dynamic memory allocation and shows how to build data structures that adapt to your needs. **Key Memory Management Functions:**

```
// Create dynamic array - like renting space for a custom-sized container
int* create_array(int size) {
    int *array = malloc(size * sizeof(int));  // Rent memory space
    if (array == NULL) {  // Check if rental was successful
        printf("Memory allocation failed!\n");
        return NULL;
    }

    // Initialize to zero - like cleaning your new apartment
    for (int i = 0; i < size; i++) {
        array[i] = 0;
    }

    return array;  // Return the address of our rented space
}

// Resize array - like moving to a bigger or smaller apartment
int* resize_array(int *array, int old_size, int new_size) {
    int *new_array = realloc(array, new_size * sizeof(int));  // Request
size change
    if (new_array == NULL) {  // Check if resize was successful
        return array;  // Keep the old array if resize failed
    }

    // Initialize new elements if array got bigger
```

```
    if (new_size > old_size) {
        for (int i = old_size; i < new_size; i++) {
            new_array[i] = 0;
        }
    }

    return new_array;  // Return the address of our resized space
}
```

Critical Memory Management Rules:

- Always check if `malloc()` returns `NULL`
- Every `malloc()` needs a matching `free()`
- Never use memory after calling `free()`
- Set pointers to `NULL` after freeing for safety

Exercise: What happens if you call `free()` on the same pointer twice?

Summary: Mastering the Advanced Programming Toolkit

Key Concepts Table

Concept	What It Does	Real-World Analogy	Why It Matters
Strings	Store and manipulate text data	Books and written language	Essential for user interfaces and data processing
Command Line Arguments	Accept parameters when program starts	Professional tool settings	Makes programs flexible and scriptable
Pointers	Direct memory access and manipulation	Address book for finding data	Enables advanced programming techniques
Dynamic Memory	Create data structures of any size	Renting apartments as needed	Efficient memory usage and flexible programs

The Professional Programming Mindset

These advanced concepts represent the transition from basic programming to professional software development. Strings enable programs to work with human language naturally. Command line arguments make programs behave like professional tools that can be customized and automated. Pointers unlock the true power of the computer's memory system, enabling efficient data manipulation and advanced algorithms. Dynamic memory management allows programs to adapt to any size problem efficiently.

The Power Progression:

- **Basic Programming**: Fixed-size variables and simple input/output
- **Intermediate Programming**: Functions, arrays, and structured data
- **Advanced Programming**: Dynamic memory, pointers, and flexible interfaces

- **Professional Programming**: Complex data structures, system integration, and optimized performance

Memory as a Fundamental Resource

Understanding memory management is like learning to manage your finances - it requires discipline, planning, and attention to detail. Every piece of memory you allocate is like money you borrow - you must pay it back (free it) when you're done. Good memory management prevents "memory leaks" (like money disappearing from your account) and keeps your programs running efficiently.

Text as Universal Communication

Strings bridge the gap between human language and computer processing. Every user interface, every file format, every network protocol involves text processing. Learning to manipulate strings effectively is like learning to speak the computer's version of human language fluently.

Command Line Interfaces as Professional Tools

Professional programmers create tools that other programmers use. Command line arguments make your programs behave like professional development tools - configurable, scriptable, and efficient. It's the difference between creating a toy and creating a tool that could be part of a professional workflow.

Final Question: How do these advanced concepts change what kinds of programs you can create? What new possibilities do they open up?

These concepts form the foundation for systems programming, game development, embedded systems, network programming, and countless other advanced applications. They transform you from someone who can write simple programs to someone who can build sophisticated software systems that solve real-world problems efficiently and elegantly.