Functions - Function definition, Function call, Function prototype,

Parameter passing; Recursion; Passing array to function; Macros - Defining and calling macros; Command line Arguments

teach me all this with appropriate points, differences to note, syntax and an example

Here's a structured explanation of functions, macros, and command-line arguments in C:

1. Functions

Function Prototype

- **Purpose**: Declares the function's name, return type, and parameters *before* its definition.
- **Syntax**:

```
return_type function_name(parameter_list);
```

- **Example**:

```
int add(int a, int b); // Prototype
```

Function Definition

- **Purpose**: Implements the function's logic.
- **Syntax**:

```
return_type function_name(parameter_list) {
    // Code
}
```

- **Example**:

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

Function Call

- **Syntax**:

  ```
  function_name(arguments);
  ```

- **Example**:

  ```
  int result = add(5, 3); // Returns 8
  ```

Parameter Passing

1. **Pass by Value** (default):
   - A copy of the argument is passed.
   - Changes inside the function do not affect the original variable.

   ```
   void increment(int x) { x++; } // Original variable unchanged
   ```

2. **Pass by Reference** (using pointers):
   - Pass the address of the variable.
   - Changes inside the function affect the original variable.

   ```
   void increment(int *x) { (*x)++; } // Original variable modified
   ```

Recursion

- A function that calls itself.
- **Example** (Factorial):

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

Passing Arrays to Functions

- Arrays decay to pointers when passed to functions.
- **Syntax**:

  ```
  void printArray(int arr[], int size); // Equivalent to int *arr
  ```

- **Example**:

```
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) printf("%d ", arr[i]);
}
```

2. Macros

Defining Macros

- Preprocessor directive for text substitution.
- **Syntax**:

  ```
  #define MACRO_NAME replacement_text
  ```

- **Example**:

  ```
  #define PI 3.14159
  #define SQUARE(x) ((x) * (x)) // Parentheses prevent errors
  ```

Calling Macros

- **Syntax**:

  ```
  MACRO_NAME(arguments);
  ```

- **Example**:

  ```
  double area = PI * SQUARE(5); // Expands to 3.14159 * (5 * 5)
  ```

Key Differences: Macros vs Functions

Feature	Macros	Functions
Type Safety	No type checking	Enforces type checking
Overhead	No runtime overhead (preprocessed)	Runtime overhead (call stack)
Debugging	Harder (text substitution)	Easier
Recursion	Not possible	Possible

3. Command-Line Arguments

Syntax for `main()`

```
int main(int argc, char *argv[]) { ... }
```

- `argc`: Number of arguments (including the program name).
- `argv[]`: Array of strings storing arguments.
    - `argv` = Program name.
    - `argv[^1]` to `argv[argc-1]` = User-provided arguments.

Example: Adding Two Numbers

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <num1> <num2>\n", argv[^0]);
        return 1;
    }
    int a = atoi(argv[^1]); // Convert string to int
    int b = atoi(argv[^2]);
    printf("Sum: %d\n", a + b);
    return 0;
}
```

Usage:

```
./program 10 20  # Output: Sum: 30
```

Key Takeaways

- **Functions**:
    - Use prototypes for forward declaration.
    - Pass arrays as pointers or with explicit size.
    - Recursion requires a base case to terminate.
- **Macros**:
    - Use parentheses to avoid precedence issues.
    - Prefer inline functions for complex logic.
- **Command-Line Arguments**:
    - Always validate `argc` before accessing `argv[]`.

    - Use `atoi()`/`atof()` to convert numeric arguments.

⁂