
Date	Dec 16 2024	Unit	1-4

Topic : C Programming

Overview of C Language

History of C

- **Developed by Dennis Ritchie** in 1972 at Bell Laboratories.
- Evolved from two earlier languages: **ALGOL** and **B** (which was created by Ken Thompson).
- Initially developed for system programming, especially for developing the UNIX operating system.
- Known for its efficiency and portability, making it widely used for embedded systems, operating systems, and application development.

Character Set

The character set in C consists of:

1. **Letters**: A–Z, a–z
2. **Digits**: 0–9
3. **Special Characters**:
   ~ `+`, `-`, `*`, `/`, `=`, `<`, `>`, `#`, etc.

i. **White Spaces**:
~ Space, Tab, Newline, etc.

Example:

```
char letter = 'A';  // Letter

int number = 123;   // Digits
```

C Tokens

Tokens are the smallest elements of a C program.
Types of tokens:

1. **Keywords**: Predefined reserved words.
2. **Identifiers**: User-defined names.
3. **Constants**: Fixed values.
4. **Strings**: Sequence of characters.
5. **Operators**: Symbols for operations.
6. **Special Symbols**: `{ }`, `[ ]`, `,`, etc.

Identifiers

- Names used to identify variables, functions, arrays, etc.
- **Rules**:
    1. Must begin with a letter or an underscore (_).
    2. Cannot be a keyword.
    3. Can contain letters, digits, and underscores.
    4. Case-sensitive.

Example:

```
int age;   // Valid identifier

int 1age;  // Invalid identifier
```

Keywords

Predefined reserved words that have special meanings.
Examples:
`int`, `float`, `if`, `else`, `return`, `void`, etc.

Data Types

1. **Basic Data Types**:
    - `int`, `float`, `double`, `char`
2. **Derived Data Types**:
    - `array`, `pointer`, `structure`, `union`
3. **Void Data Type**:

- `void` (for functions with no return type)

Example:

```
int age = 25;

float salary = 5000.50;

char grade = 'A';
```

Variables

- Used to store data.
- **Declaration**: Specifies the type and name.
- **Initialization**: Assigns a value.

Example:

```
int num = 10;  // Declaration and initialization
```

Constants

Fixed values that cannot be modified during the program's execution.
Types:

1. **Integer Constants**: `10`, `-20`
2. **Floating-point Constants**: `3.14`, `-0.99`
3. **Character Constants**: `'A'`, `'@'`
4. **String Constants**: `"Hello"`

Symbolic Constants

- Named constants defined using `#define`.

Example:

```
#define PI 3.14
```

Operators in C

Categories of Operators:

1. **Arithmetic Operators**: `+`, `-`, `*`, `/`, `%`
2. **Relational Operators**: `>`, `<`, `==`, `!=`, `>=`, `<=`

3. **Logical Operators**: `&&`, `||`, `!`
4. **Bitwise Operators**: `&`, `|`, `^`, `~`, `<<`, `>>`
5. **Assignment Operators**: `=`, `+=`, `-=`, etc.
6. **Increment/Decrement Operators**: `++`, `--`
7. **Conditional Operator**: `? :`

Syntax

```
condition ? value_if_true : value_if_false;
```

Example Program

```
#include <stdio.h>

int main() {

    int a = 10, b = 20;

    int max;

    // Using the conditional operator to find the maximum value

    max = (a > b) ? a : b;

    printf("The maximum value is: %d\n", max);

    return 0;

}
```

Explanation

1. `(a > b)` is the condition.
2. If `a > b` is true, the value of `a` is returned.
3. If `a > b` is false, the value of `b` is returned.

Output

```
The maximum value is: 20
```

Hierarchy of Operators

Operator precedence determines the order in which operators are evaluated.

1. Parentheses `( )`
2. Unary Operators `!`, `++`, `--`
3. Multiplicative Operators `*`, `/`, `%`
4. Additive Operators `+`, `-`
5. Relational Operators `<`, `>`
6. Logical Operators `&&`, `||`
7. Assignment Operators `=`, `+=`, `-=`

Example:

```
int result = 10 + 20 * 5;  // Multiplication (*) evaluated before
addition (+)
```

Expressions

- A combination of variables, constants, and operators.

Example:

```
int a = 5, b = 10;

int sum = a + b;  // Expression: a + b
```

Type Conversions

1. **Implicit Type Conversion** (Type Promotion):
   Smaller data types are automatically converted to larger types.
   Example: `int` → `float`

Explicit Type Conversion (Type Casting):
Forcefully converting one data type to another.
Example:

```
float num = (float) 5 / 2;  // Result: 2.5
```

Library Functions

- **Math Functions**: `sqrt()`, `pow()`, `sin()`
- **String Functions**: `strcpy()`, `strlen()`
- **Input/Output Functions**: `printf()`, `scanf()`

Managing Input and Output Operations

Formatted I/O Functions

Used for input/output with format specifiers:

`printf()`: Outputs data to the console.
Syntax:

```
printf("Format String", variable1, variable2);
```

Example:

```
int age = 25;

printf("Age is %d", age);  // Output: Age is 25
```

1. **`scanf()`**: Inputs data from the user.
   Syntax:

   ```
   scanf("Format String", &variable);
   ```

Example:

```
int num;

scanf("%d", &num);  // User inputs an integer value
```

Unformatted I/O Functions

Used for raw input/output:

`getchar()`: Reads a single character.
Example:

```
char ch;

ch = getchar();
```

1. **`putchar()`**: Outputs a single character.
   Example:

```
putchar('A');
```

2. **`gets()`**: Reads a string.
   Example:

```
char str[50];

gets(str);
```

3. **`puts()`**: Outputs a string.
   Example:

```
char str[] = "Hello";

puts(str);
```

Example Program

```
#include <stdio.h>

int main() {

    int age;

    float salary;

    printf("Enter your age: ");

    scanf("%d", &age);  // Input age

    printf("Enter your salary: ");

    scanf("%f", &salary);  // Input salary

    printf("Age: %d, Salary: %.2f\n", age, salary);  // Display values

    return 0;

}
```

Decision Making, Branching, and Looping

Decision Making Statements

Decision-making statements allow the program to execute certain sections of code based on conditions.

1. `if` Statement

- Executes a block of code only if the condition is `true`.

Syntax:

```
if (condition) {
    // code to execute if condition is true
}
```

Example:

```
int age = 18;
if (age >= 18) {
    printf("You are eligible to vote.\n");
}
```

2. `if-else` Statement

- Provides two paths of execution based on whether the condition is `true` or `false`.

Syntax:

```
if (condition) {
    // code to execute if condition is true
} else {
    // code to execute if condition is false
}
```

Example:

```
int num = 5;
if (num % 2 == 0) {
    printf("Even number.\n");
} else {
    printf("Odd number.\n");
}
```

3. Nested `if` Statement

- An `if` statement inside another `if` statement.

Syntax:

```
if (condition1) {
    if (condition2) {
        // code to execute if both conditions are true
    }
}
```

Example:

```
int marks = 85;
if (marks > 50) {
    if (marks >= 85) {
        printf("Excellent performance.\n");
    }
}
```

4. `else-if` Ladder

- Allows checking multiple conditions sequentially.

Syntax:

```
if (condition1) {
    // code for condition1
} else if (condition2) {
    // code for condition2
} else {
    // code if none of the conditions are true
}
```

Example:

```
int num = 0;
if (num > 0) {
    printf("Positive number.\n");
} else if (num < 0) {
```

```
    printf("Negative number.\n");
} else {
    printf("Zero.\n");
}
```

5. `switch` Statement

- Used to select one of many blocks of code to execute.

Syntax:

```
switch (expression) {
    case value1:
        // code for value1
        break;
    case value2:
        // code for value2
        break;
    default:
        // code if no case matches
}
```

Example:

```
int choice = 2;
switch (choice) {
    case 1:
        printf("You selected option 1.\n");
        break;
    case 2:+
        printf("You selected option 2.\n");
        break;
    default:
        printf("Invalid option.\n");
}
```

Looping

Loops are used to execute a block of code multiple times.

1. `while` Loop

- Repeats as long as the condition is true.

Syntax:

```
while (condition) {
    // code to execute
}
```

Example:

```
int i = 1;
while (i <= 5) {
    printf("%d\n", i);
    i++;
}
```

2. `do-while` Loop

- Executes at least once, then repeats as long as the condition is true.

Syntax:

```
do {
    // code to execute
} while (condition);
```

Example:

```
int i = 1;
do {
    printf("%d\n", i);
    i++;
} while (i <= 5);
```

3. `for` Loop

- Executes a block of code a specified number of times.

Syntax:

```
for (initialization; condition; increment/decrement) {
    // code to execute
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
    printf("%d\n", i);
}
```

4. Nested Loops

- A loop inside another loop.

Example:

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        printf("i = %d, j = %d\n", i, j);
    }
}
```

5. `break` Statement

- Terminates the loop or switch statement.

Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) break;
    printf("%d\n", i);
}
```

6. `continue` Statement

- Skips the current iteration and moves to the next iteration.

Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    printf("%d\n", i);
```

```
}
```

7. `goto` Statement

- Transfers control to a labeled statement.

Example:

```
int i = 1;
start:
    printf("%d\n", i);
    i++;
    if (i <= 5) goto start;
```

Functions

Functions are reusable blocks of code that perform a specific task.

1. Function Definition

- **Syntax**:

```
return_type function_name(parameters) {
    // body of the function
}
```

Example:

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

2. Function Prototyping

- Declares the function before its definition.

Syntax:

```
return_type function_name(parameters);
```

Example:

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

3. Types of Functions

1. **Built-in Functions**: Provided by C libraries, e.g., `printf()`, `scanf()`.
2. **User-defined Functions**: Created by the programmer.

4. Passing Arguments to Functions

Pass by Value: Copies the value of arguments. Example:

```
void display(int num) {
    printf("%d\n", num);
}
```

Pass by Reference: Passes the address of arguments. Example:

```
void modify(int *num) {
    *num = 10;
}
```

5. Nested Functions

C does not directly support nested functions, but they can be mimicked using function calls within other functions.

Example:

```
void outer() {
    printf("Outer function.\n");
    inner();  // Call to another function
}
void inner() {
    printf("Inner function.\n");
}
```

6. Recursive Functions

- A function that calls itself.

Example:

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

Example Program Using Functions

```
#include <stdio.h>

// Function prototype
int factorial(int);

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

// Function definition
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

Arrays

An array is a collection of elements of the same data type stored in contiguous memory locations.

1. Declaring and Initializing Arrays

- **Declaration**:

```
data_type array_name[size];
```

- **Initialization**:

```
int arr[5] = {1, 2, 3, 4, 5};  // Static initialization
int arr[5] = {0};              // Initializes all elements to 0
```

Example:

```
int arr[3];
```

```
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
```

2. One-Dimensional Arrays

- A single row of elements.
- **Example**:

```
#include <stdio.h>
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

3. Two-Dimensional Arrays

- Represents data in a table format (rows and columns).
- **Declaration**:

```
data_type array_name[rows][columns];
```

- **Initialization**:

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

Example:

```
#include <stdio.h>
int main() {
    int arr[2][2] = {{1, 2}, {3, 4}};
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
```

```
    return 0;
}
```

4. Multi-Dimensional Arrays

- Arrays with more than two dimensions.
- **Example**:

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

5. Passing Arrays to Functions

- **Syntax**:

```
void function_name(data_type array_name[], int size);
```

Example:

```
#include <stdio.h>
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
}
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    printArray(arr, 5);
    return 0;
}
```

Strings

A string is a one-dimensional array of characters, terminated by a null character (`\0`).

1. Declaring and Initializing Strings

- **Declaration**:

```
char str[size];
```

- **Initialization**:

```
char str[] = "Hello";    // Automatically adds '\0'
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

2. Operations on Strings

Common string operations are provided by the `<string.h>` library.

- **Length**:

```
#include <string.h>
int len = strlen("Hello");
```

- **Copy**:

```
strcpy(destination, source);
```

- **Concatenate**:

```
strcat(str1, str2);
```

- **Compare**:

```
strcmp(str1, str2);  // Returns 0 if equal
```

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[20] = "Hello";
    char str2[20] = "World";
    strcat(str1, str2);
    printf("%s\n", str1);
    return 0;
}
```

3. Arrays of Strings

- An array of strings is a 2D array where each row represents a string.
- **Example**:

```
char fruits[3][10] = {"Apple", "Banana", "Cherry"};
```

4. Passing Strings to Functions

- **Example**:

```
#include <stdio.h>
void printString(char str[]) {
    printf("String: %s\n", str);
}
int main() {
    char str[] = "Hello";
    printString(str);
    return 0;
}
```

Storage Classes

Storage classes define the scope, visibility, and lifetime of variables in a program.

1. Automatic (`auto`)

- Default storage class for local variables.
- **Example**:

```
int main() {
    auto int a = 10;  // Same as int a = 10;
    printf("%d\n", a);
    return 0;
}
```

2. External (`extern`)

- Declares a global variable in another file.
- **Example**:

```
extern int a;
```

The `extern` keyword in C is used to declare a global variable or function in another file or scope. It tells the compiler that the variable is defined elsewhere, so it does not allocate memory for it again but links to the definition.

Here's an example program demonstrating `extern int a`:

File 1: `file1.c`

This file contains the definition of the variable `a`:

```
#include <stdio.h>

// Defining the variable 'a'
int a = 10;

void display() {
    printf("Value of a in file1.c: %d\n", a);
}
```

File 2: `file2.c`

This file uses the `extern` keyword to declare `a`:

```
#include <stdio.h>

// Declaring 'a' as an external variable
extern int a;

void display();

int main() {
    printf("Value of a in file2.c: %d\n", a);  // Accessing the
external variable
    a = 20;  // Modifying the external variable
    display();  // Calling the function from file1.c
    return 0;
}
```

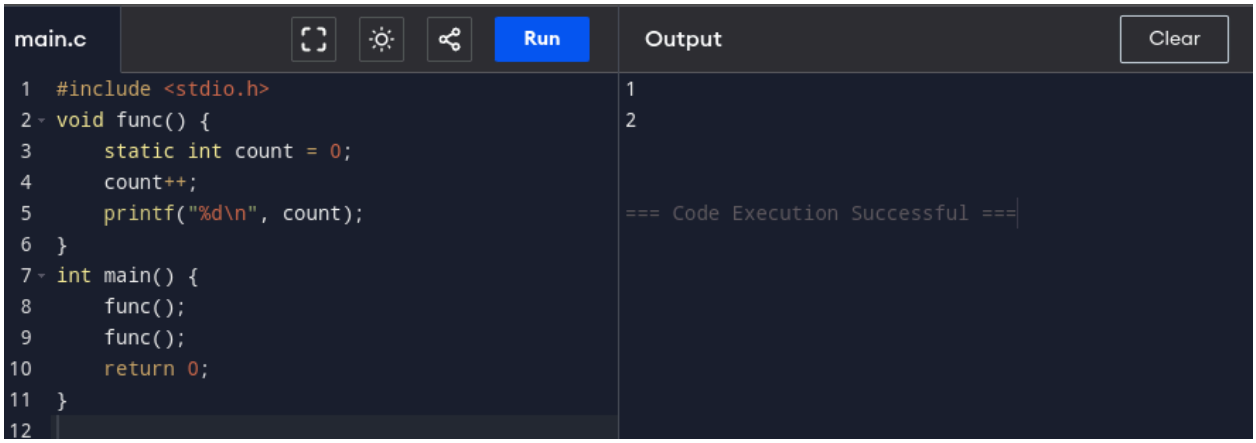
Steps to Compile and Run

1. Compile both files together:
   `gcc file1.c file2.c -o extern_example`
2. Run the executable:
   `./extern_example`

3. Static

- Preserves the value of a variable between function calls.
- **Example**:

```
#include <stdio.h>
void func() {
    static int count = 0;
    count++;
    printf("%d\n", count);
}
int main() {
    func();
    func();
    return 0;
}
```



4. Register

- Stores variables in CPU registers for faster access.
- **Example**:

```
register int a = 5;
```

In C, the `register` keyword is used to suggest to the compiler that the variable should be stored in a CPU register instead of RAM for faster access. However, this is only a suggestion,

and modern compilers may ignore it based on optimization needs. Additionally, you cannot take the address of a `register` variable using the address-of operator (&).

main.c Run Output Clear

```c
#include <stdio.h>

int main() {
    register int a = 5;  // Declare a register variable

    printf("The value of a is: %d\n", a);

    // Uncommenting the following line will cause a compile error
    // because the address of a register variable cannot be taken.
    printf("Address of a: %p\n", &a);

    for (register int i = 0; i < 5; i++) {  // Using register in a loop variable
        printf("Loop iteration: %d\n", i);
    }

    return 0;
}
```

```
/tmp/PqjUC4x5NC/main.c: In function 'main':
ERROR!
/tmp/PqjUC4x5NC/main.c:10:5: error: address of register variable 'a' requested
   10 |     printf("Address of a: %p\n", &a);
      |     ^~~~~~

=== Code Exited With Errors ===
```

main.c Run Output Clear

```c
#include <stdio.h>

int main() {
    register int a = 5;  // Declare a register variable

    printf("The value of a is: %d\n", a);

    // Uncommenting the following line will cause a compile error
    // because the address of a register variable cannot be taken.
    //printf("Address of a: %p\n", &a);

    for (register int i = 0; i < 5; i++) {  // Using register in a loop variable
        printf("Loop iteration: %d\n", i);
    }

    return 0;
}
```

```
The value of a is: 5
Loop iteration: 0
Loop iteration: 1
Loop iteration: 2
Loop iteration: 3
Loop iteration: 4

=== Code Execution Successful ===
```

Structures

A structure is a user-defined data type that groups related variables.

1. Declaring and Initializing Structures

- **Declaration**:

```
struct structure_name {
    data_type member1;
    data_type member2;
};
```

- **Initialization**:

```
struct structure_name var = {value1, value2};
```

Example:

```
#include <stdio.h>
struct Point {
    int x;
    int y;
};
int main() {
    struct Point p = {10, 20};
    printf("x = %d, y = %d\n", p.x, p.y);
    return 0;
}
```

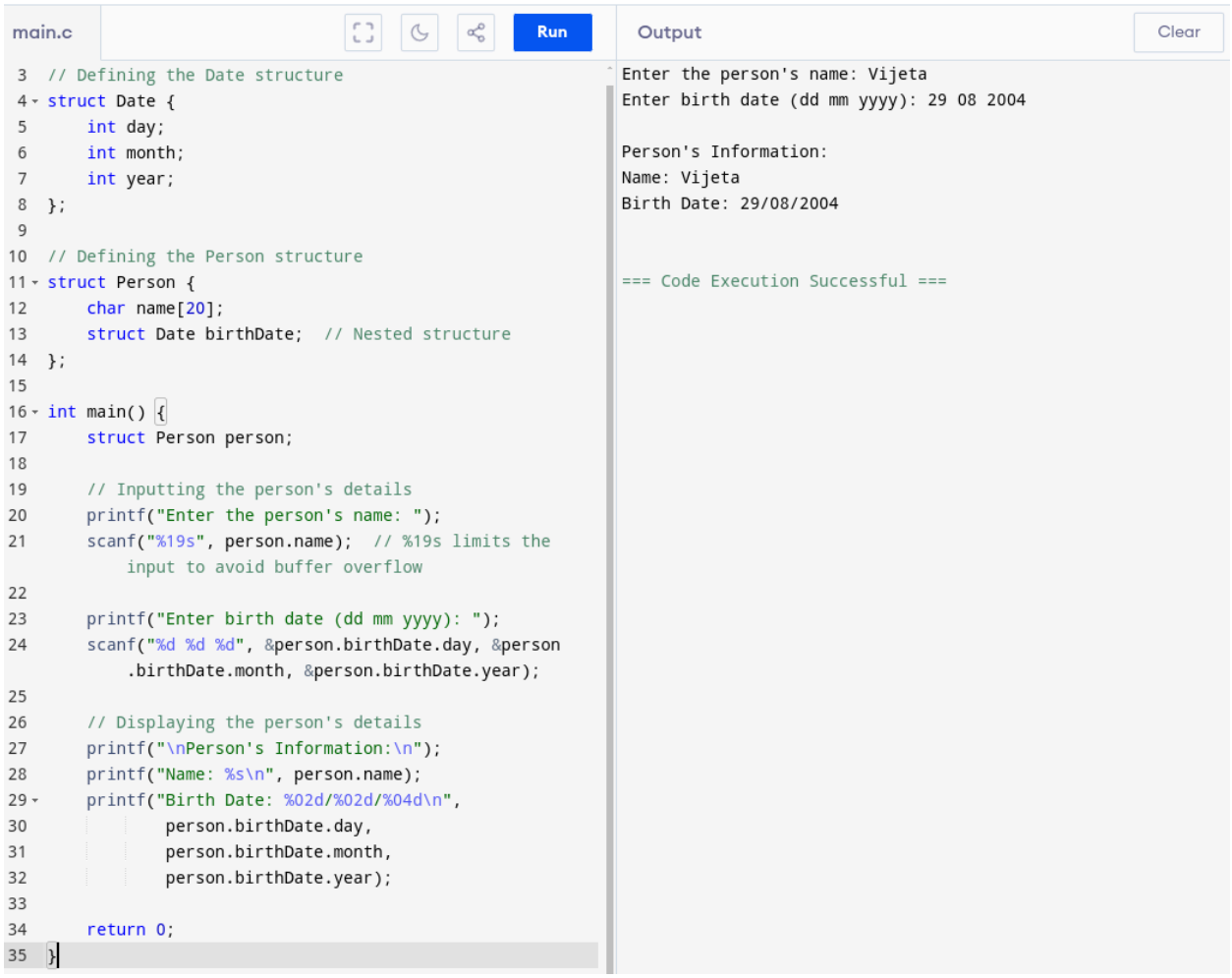
2. Nested Structures

- A structure inside another structure.

Example:

```
struct Date {
    int day;
    int month;
```

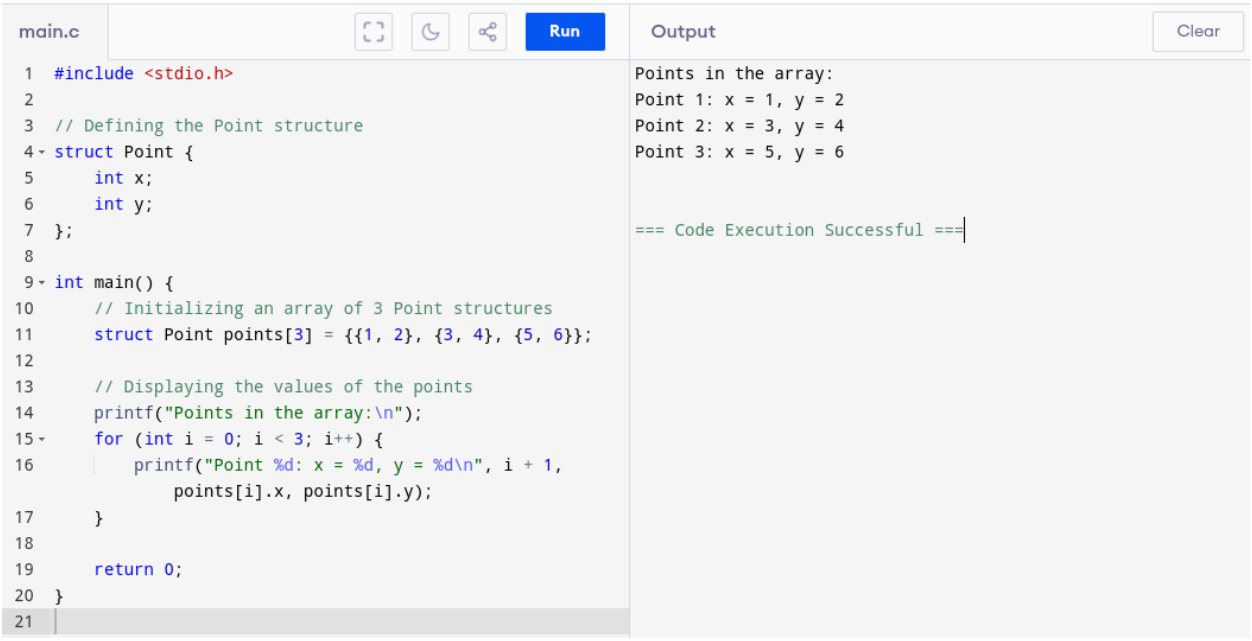
```
    int year;
};
struct Person {
    char name[20];
    struct Date birthDate;
};
```



3. Array of Structures

- **Example**:

```
struct Point {
    int x;
    int y;
};
struct Point points[3] = {{1, 2}, {3, 4}, {5, 6}};
```



4. Passing Structures to Functions

- **Example**:

```
void display(struct Point p) {
    printf("x = %d, y = %d\n", p.x, p.y);
}
```

main.c

```c
#include <stdio.h>

// Defining the Point structure
struct Point {
    int x;
    int y;
};

// Function to display a point
void display(struct Point p) {
    printf("x = %d, y = %d\n", p.x, p.y);
}

int main() {
    // Initializing an array of Point structures
    struct Point points[3] = {{1, 2}, {3, 4}, {5, 6}};

    printf("Displaying points:\n");
    // Calling the display function for each point
    for (int i = 0; i < 3; i++) {
        printf("Point %d: ", i + 1);
        display(points[i]);
    }

    return 0;
}
```

Output

```
Displaying points:
Point 1: x = 1, y = 2
Point 2: x = 3, y = 4
Point 3: x = 5, y = 6


=== Code Execution Successful ===
```

Unions

- Similar to structures, but members share the same memory space.

Example:

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

Typedef

- Creates an alias for a data type.

Example:

```
typedef unsigned int uint;
uint a = 5;
```

Enumerations (`enum`)

- Defines a set of named integral constants.

Example:

```
enum Colors {RED, GREEN, BLUE};
enum Colors color = RED;
```

Bit Fields

- Used to store data in a compact form within a structure.

Example:

```
struct {
    unsigned int age : 3;  // Uses only 3 bits
} person;
```

Pointers

Pointers are variables that store the memory address of another variable.

1. Declaration and Initialization

- **Syntax**:

```
data_type *pointer_name;
```

- **Example**:

```
int a = 10;
int *p = &a;  // Pointer p stores the address of variable a
```

2. Pointer Arithmetic

- Operations include increment (`++`), decrement (`--`), addition (`+`), and subtraction (`-`).
- **Example**:

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

```
int main() {
    int arr[] = {10, 20, 30};
    int *p = arr;
    printf("%d\n", *p);       // Prints 10
    p++;
    printf("%d\n", *p);       // Prints 20
    return 0;
}
```

3. Pointers and Functions

- Passing pointers to functions allows direct modification of variables.

Call by Value: A copy of the variable is passed, and changes are not reflected.

```
void modify(int a) {
    a = 20;
}
```

Call by Reference: A pointer is passed, and changes are reflected.

```
void modify(int *p) {
    *p = 20;
}
```

4. Pointers and Arrays

- A pointer can point to the first element of an array.
- **Example**:

```
#include <stdio.h>
int main() {
    int arr[] = {1, 2, 3};
    int *p = arr;
    for (int i = 0; i < 3; i++) {
        printf("%d ", *(p + i));  // Accessing array elements using
pointers
    }
    return 0;
}
```

5. Arrays of Pointers

- Array elements can be pointers.
- **Example**:

```
#include <stdio.h>
int main() {
    char *arr[] = {"Hello", "World"};
    printf("%s %s\n", arr[0], arr[1]);
    return 0;
}
```

6. Pointers and Structures

- **Example**:

```
#include <stdio.h>
struct Point {
    int x, y;
};
int main() {
    struct Point p = {10, 20};
    struct Point *ptr = &p;
    printf("x = %d, y = %d\n", ptr->x, ptr->y);  // Using arrow
operator
    return 0;
}
```

Memory Allocation

Memory can be allocated at runtime using dynamic memory allocation.

1. Static vs Dynamic Memory Allocation

- **Static**: Memory is allocated at compile time.
- **Dynamic**: Memory is allocated at runtime using library functions.

2. Memory Allocation Functions

- `malloc` (Memory Allocation): Allocates uninitialized memory.

- `calloc` (Contiguous Allocation): Allocates zero-initialized memory.
- `realloc`: Resizes previously allocated memory.
- `free`: Deallocates dynamically allocated memory.

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *ptr = (int *)malloc(5 * sizeof(int));  // Allocating memory
for 5 integers
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        return 1;
    }
    for (int i = 0; i < 5; i++) {
        ptr[i] = i + 1;
        printf("%d ", ptr[i]);
    }
    free(ptr);  // Deallocating memory
    return 0;
}
```

Files

Files in C are used for data storage and retrieval.

1. File Modes

- `"r"`: Open for reading.
- `"w"`: Open for writing (overwrites existing content).
- `"a"`: Open for appending.

2. File Operations

- **File Functions**:
    - `fopen`: Opens a file.
    - `fclose`: Closes a file.
    - `fprintf`/`fscanf`: Formatted I/O.
    - `fgets`/`fputs`: Reads/writes strings.

- `fread`/`fwrite`: Reads/writes binary data.

Example:

```
#include <stdio.h>
int main() {
    FILE *fp = fopen("example.txt", "w");
    if (fp == NULL) {
        printf("Error opening file.\n");
        return 1;
    }
    fprintf(fp, "Hello, File!\n");
    fclose(fp);
    return 0;
}
```

3. Text and Binary Files

- **Text File**: Data is stored in human-readable form.
- **Binary File**: Data is stored in binary format.

Binary File Example:

```
#include <stdio.h>
struct Data {
    int id;
    char name[20];
};
int main() {
    struct Data d = {1, "Test"};
    FILE *fp = fopen("data.bin", "wb");
    fwrite(&d, sizeof(struct Data), 1, fp);
    fclose(fp);
    return 0;
}
```

Command Line Arguments

Command-line arguments are passed to `main` as arguments.

Syntax:

```
int main(int argc, char *argv[]) {
    // argc: Argument count
    // argv: Argument vector (array of strings)
}
```

Example:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }
    return 0;
}
```

C Preprocessor Directives

Preprocessor directives are commands processed before compilation.

1. Common Directives

- `#include`: Includes header files.
- `#define`: Defines macros.
- `#ifdef`/`#ifndef`: Conditional compilation.

Example:

```
#include <stdio.h>
#define PI 3.14
int main() {
    printf("PI = %.2f\n", PI);
    return 0;
}
```

Macros

A macro is a fragment of code defined using `#define`.

1. Types of Macros

- **Object-like Macros**:

```
#define MAX 100
```

- **Function-like Macros**:

```
#define SQUARE(x) ((x) * (x))
```

Example:

```
#include <stdio.h>
#define SQUARE(x) ((x) * (x))
int main() {
    printf("Square of 5: %d\n", SQUARE(5));
    return 0;
}
```

User-Defined Header Files

Custom headers can be created for modularity.

Steps:

1. Create a file (`myheader.h`):

   ```
   #define HELLO "Hello, World!"
   ```

2. Include it in your program:

   ```
   #include "myheader.h"
   #include <stdio.h>
   int main() {
       printf("%s\n", HELLO);
       return 0;

   }
   ```

Programs to try

Advanced: Pointer Arithmetic

Problem: Write a program to find the sum of an array using pointers.

```
#include <stdio.h>

int sumArray(int *arr, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += *(arr + i);
    }
    return sum;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Sum of the array: %d\n", sumArray(arr, size));

    return 0;
}
```

Switch Statement: Simple Calculator

Problem: Write a program to implement a simple calculator using a switch statement.

```
#include <stdio.h>

int main() {
    char operator;
    double num1, num2;

    printf("Enter operator (+, -, *, /): ");
    scanf("%c", &operator);
    printf("Enter two numbers: ");
    scanf("%lf %lf", &num1, &num2);
```

```
    switch (operator) {
        case '+':
            printf("Result: %.2lf\n", num1 + num2);
            break;
        case '-':
            printf("Result: %.2lf\n", num1 - num2);
            break;
        case '*':
            printf("Result: %.2lf\n", num1 * num2);
            break;
        case '/':
            if (num2 != 0)
                printf("Result: %.2lf\n", num1 / num2);
            else
                printf("Error: Division by zero is not allowed.\n");
            break;
        default:
            printf("Invalid operator.\n");
    }

    return 0;
}
```

Pointer to Array: Access Elements

Problem: Write a program to access elements of an array using a pointer.

```
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr;

    printf("Array elements using pointer:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", *(ptr + i));
    }
```

```
    return 0;
}
```

Strings: Count Vowels in a String

Problem: Write a program to count the number of vowels in a given string.

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

int countVowels(char str[]) {
    int count = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        char ch = str[i];
        if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch ==
'u' ||
            ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch ==
'U') {
            count++;
        }
    }
    return count;
}

int main() {
    char str[100];
    printf("Enter a string: ");
    gets(str); // Use fgets(str, sizeof(str), stdin) in modern C for
safety.
    printf("Number of vowels: %d\n", countVowels(str));

    return 0;
}
```

Pointers: Swap Two Numbers Using Pointers

Problem: Write a program to swap two numbers using pointers.

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

```
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 10, b = 20;
    printf("Before Swap: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After Swap: a = %d, b = %d\n", a, b);

    return 0;
}
```

2D Array: Matrix Addition

Problem: Write a program to add two 2x2 matrices.

```
#include <stdio.h>

void addMatrices(int a[2][2], int b[2][2]) {
    int result[2][2];
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            result[i][j] = a[i][j] + b[i][j];
        }
    }

    printf("Resultant Matrix:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }
}

int main() {
```

```
    int a[2][2] = {{1, 2}, {3, 4}};
    int b[2][2] = {{5, 6}, {7, 8}};
    addMatrices(a, b);

    return 0;
}
```

Array: Reverse an Array

Problem: Write a program to reverse an array of integers.

```
#include <stdio.h>

void reverseArray(int arr[], int size) {
    printf("Reversed Array: ");
    for (int i = size - 1; i >= 0; i--) {
        printf("%d ", arr[i]);
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    reverseArray(arr, size);

    return 0;
}
```

Assignment

Arrays

1. Write a program to **input and print** 5 numbers using an array.
2. Write a program to find the **largest number** in an array of 5 integers.
3. Write a program to calculate the **sum of all elements** in an array.
4. Store 10 integers in an array and **print only the even numbers**.
5. Write a program to count how many times the number 5 appears in an array of 10 elements.

2D Arrays

1. Write a program to **input and print** a 2x2 matrix.
2. Write a program to find the **sum of all elements** in a 2x2 matrix.
3. Write a program to input a 2x2 matrix and print the **elements in row-major order**.
4. Write a program to find the **largest number** in a 2x2 matrix.
5. Write a program to **input two 2x2 matrices** and print their addition.

Pointers

1. Write a program to print the **value and address** of an integer variable using a pointer.
2. Write a program to swap two numbers using pointers.
3. Write a program to increment a variable by 10 using a pointer.
4. Write a program to display the elements of an array using a pointer.
5. Write a program to calculate the sum of two numbers using pointers.

Strings

1. Write a program to **input and print a string**.
2. Write a program to find the **length of a string** without using the `strlen()` function.
3. Write a program to **convert a string to uppercase** (use only basic logic, not inbuilt functions).
4. Write a program to **compare two strings** and print if they are the same or different.
5. Write a program to **count the number of vowels** in a string.

Switch Statements

1. Write a program using a switch statement to display the **day of the week** based on user input (1 for Monday, 2 for Tuesday, etc.).
2. Write a program using a switch statement to check if a character is a **vowel or consonant**.
3. Write a simple calculator program using a switch statement (support `+`, `-`, `*`, `/`).
4. Write a program using a switch statement to print the **month name** based on the number (1 for January, 2 for February, etc.).
5. Write a program using a switch statement to check if a given number is **positive, negative, or zero**.

Level 2 Questions

1. Write a program to **find the smallest number** in an array of 5 elements.
2. Write a program to input a string and print it **character by character** using a loop.
3. Write a program to calculate the **sum of diagonal elements** of a 2x2 matrix.
4. Write a program to print the **ASCII values** of each character in a string.
5. Write a program to find whether a number is **even or odd** using a switch statement.

END