CAVALIER INSTITUTIONS
AFFILIATED TO BANGALORE NORTH UNIVERSITY / GOVERNMENT OF KARNATAKA

Cavalier Institute - https://cavalierinstitutions.com

Remaining topics

`continue` Statement

The `continue` statement skips the rest of the loop iteration and moves to the next iteration.

main.c

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            continue; // Skip the rest of the loop for even numbers
        }
        printf("%d ", i); // Print only odd numbers
    }
    return 0;
}
```

Output

```
1 3 5 7 9

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

Recursive Functions

A recursive function calls itself. Below is an example of calculating the factorial of a number.

main.c

```c
#include <stdio.h>

// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1; // Base case
    }
    return n * factorial(n - 1); // Recursive call
}

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

Output

```
Factorial of 5 is 120


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

String Functions Examples

C provides functions in `<string.h>` for string manipulation. Below is an example using `strlen`, `strcpy`, and `strcmp`.

main.c

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

int main() {
    char str1[50] = "Hello";
    char str2[50];

    // Copy str1 to str2
    strcpy(str2, str1);
    printf("Copied String: %s\n", str2);

    // Find length of str1
    printf("Length of str1: %d\n", strlen(str1));

    // Compare strings
    if (strcmp(str1, str2) == 0) {
        printf("Strings are equal\n");
    } else {
        printf("Strings are not equal\n");
    }

    return 0;
}
```

Output

```
Copied String: Hello
Length of str1: 5
Strings are equal


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

Enum Example

An `enum` (short for **enumeration**) is a user-defined data type in C that consists of integral constants. Each enumerator (or constant) is assigned an integer value, starting from `0` by default unless explicitly specified. Enums are primarily used to make code more readable and maintainable by using meaningful names instead of plain numbers.

How Enums Work

1. An `enum` defines a set of named constants that represent integer values.

enum Color { RED, GREEN, BLUE };

Here:

- `RED` = `0`
- `GREEN` = `1`
- `BLUE` = `2`

You can assign custom values to enumerators.

enum Color { RED = 1, GREEN = 5, BLUE = 10 };

Use Cases:

- Making code more readable by replacing numbers with names.
- Managing related constants, e.g., days of the week, error codes, colors, etc.

main.c

```c
#include <stdio.h>

enum ErrorCodes {
    SUCCESS = 0,
    ERROR_FILE_NOT_FOUND = 404,
    ERROR_ACCESS_DENIED = 403,
    ERROR_UNKNOWN = -1
};

int main() {
    enum ErrorCodes error = ERROR_FILE_NOT_FOUND;

    if (error == ERROR_FILE_NOT_FOUND) {
        printf("Error: File Not Found (Code: %d)\n",
            ERROR_FILE_NOT_FOUND);
    }

    return 0;
}
```

Output

```
ERROR!
Error: File Not Found (Code: 404)


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

Enum as Flags

Enums can be used to represent a set of bit flags. This is particularly useful in systems programming or when managing states.

main.c

```c
#include <stdio.h>

enum Permissions {
    READ = 1,   // 0001
    WRITE = 2,  // 0010
    EXECUTE = 4 // 0100
};

int main() {
    int userPermissions = READ | EXECUTE; // Combine permissions using bitwise OR
    // this is also 1+4 = 5

    printf("%d \n", userPermissions );

    if (userPermissions & READ) {
        printf("User has read permission.\n");
    }
    if (userPermissions & WRITE) {
        printf("User has write permission.\n");
    }
    if (userPermissions & EXECUTE) {
        printf("User has execute permission.\n");
    }

    return 0;
}
```

Output

```
5
User has read permission.
User has execute permission.


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

main.c

```c
#include <stdio.h>

enum Weekday { MON, TUE, WED, THU, FRI, SAT, SUN };

int main() {
    enum Weekday today = WED;

    if (today == WED) {
        printf("It's Wednesday!\n");
    }

    // Print all days of the week
    for (int day = MON; day <= SUN; day++) {
        printf("Day %d\n", day);
    }

    return 0;
}
```

Output

```
It's Wednesday!
Day 0
Day 1
Day 2
Day 3
Day 4
Day 5
Day 6


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

Enum with Switch Case

Enums work well with switch-case statements for clean and readable code.

main.c

```c
#include <stdio.h>

enum TrafficLight { RED, YELLOW, GREEN };

int main() {
    enum TrafficLight signal = GREEN;

    switch (signal) {
    case RED:
        printf("Stop!\n");
        break;
    case YELLOW:
        printf("Get Ready...\n");
        break;
    case GREEN:
        printf("Go!\n");
        break;
    default:
        printf("Invalid Signal\n");
    }

    return 0;
}
```

Output

```
Go!


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

`malloc` - Dynamic Memory Allocation

`malloc` is used to allocate memory dynamically. It allocates memory in the heap and returns a pointer.

1. **Dynamic Allocation**:
   - The memory for the array is allocated at runtime, not during compile time.
2. **Memory Size**:
   - If `n = 5` and `sizeof(int) = 4`, then `malloc` allocates `20 bytes` of memory.
3. **Pointer Use**:
   - `arr` points to the first element of the dynamically allocated array. You can access and modify the elements using array indexing (`arr[i]`) or pointer arithmetic (`*(arr + i)`).
4. **Freeing Memory**:
   - After usage, the memory allocated with `malloc` must be released using `free(arr)` to avoid **memory leaks**.

If you don't `free(arr)`, the allocated memory will not be reclaimed until the program terminates, leading to inefficient use of system resources.

```c
#include <stdio.h>
#include <stdlib.h> // Required for malloc and free

int main() {
    int n, *arr;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Allocate memory for n integers
    arr = (int *)malloc(n * sizeof(int));

    printf("%p\n", arr);

    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Initialize and print the array
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    printf("Array elements: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    // Free allocated memory
    free(arr);

    return 0;
}
```

Output

```
Enter the number of elements: 5
0xc10fac0
Array elements: 1 2 3 4 5

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

File operations

File operations in C allow programs to interact with files to store, retrieve, and process data. The key functions for file handling are provided in the `stdio.h` library. Common operations include reading, writing, appending, and closing files.

Key Functions

1. **`fopen`**: Opens a file.
2. **`fclose`**: Closes a file.
3. **`fprintf`**: Writes formatted data to a file.
4. **`fscanf`**: Reads formatted data from a file.
5. **`fgets`/`fputs`**: Reads/writes strings from/to a file.
6. **`fread`/`fwrite`**: Reads/writes binary data.

File Modes

- `"r"`: Open for reading.
- `"w"`: Open for writing (overwrites existing content or creates a new file).
- `"a"`: Open for appending.
- `"r+"`: Open for reading and writing.
- `"w+"`: Open for reading and writing (overwrites existing content).
- `"a+"`: Open for reading and appending.

```
files.c
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ gcc files.c -o x
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ls
files.c  x
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ./x
Data written to file successfully.
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ls
files.c  test_file.txt  x
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ vim test_file.txt
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ cat files.c
#include <stdio.h>

int main() {
    FILE *file;

    // Open file for writing
    file = fopen("test_file.txt", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // Write to the file
    fprintf(file, "Hello, World!\n");
    fprintf(file, "This is a file example in C.\n");

    // Close the file
    fclose(file);

    printf("Data written to file successfully.\n");
    return 0;
}

xbitlabsin@fedora:~/Cavalier-December/sem1/files1$
```

```
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ cat
files.c        test_file.txt  x
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ cat test_file.txt
Hello, World!
This is a file example in C.
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$
```

Explanation

`fopen("example.txt", "w")`: Opens the file in write mode. If the file doesn't exist, it creates a new one.

`fprintf`: Writes formatted text to the file.

`fclose`: Closes the file to ensure data is saved and resources are freed.

Example 2: Reading from a File

```
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ vim example2.c
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ cat example2.c
#include <stdio.h>

int main() {
    FILE *file;
    char buffer[100];

    // Open file for reading
    file = fopen("example.txt", "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // Read and display file content
    printf("File Content:\n");
    while (fgets(buffer, sizeof(buffer), file)) {
        printf("%s", buffer);
    }

    // Close the file
    fclose(file);

    return 0;
}

xbitlabsin@fedora:~/Cavalier-December/sem1/files1$
```

```c
#include <stdio.h>

int main() {
    FILE *file;
    char buffer[100];

    // Open file for reading
    file = fopen("example.txt", "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // Read and display file content
    printf("File Content:\n");
    while (fgets(buffer, sizeof(buffer), file)) {
        printf("%s", buffer);
    }

    // Close the file
    fclose(file);

    return 0;
}
```

```
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ gcc example2.c -o ex2
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ls -lrt
total 52
-rw-r--r--. 1 xbitlabsin xbitlabsin   436 Dec 18 11:23 files.c
-rwxr-xr-x. 1 xbitlabsin xbitlabsin 16768 Dec 18 11:23 x
-rw-r--r--. 1 xbitlabsin xbitlabsin    43 Dec 18 11:23 test_file.txt
-rw-r--r--. 1 xbitlabsin xbitlabsin   442 Dec 18 11:26 example2.c
-rwxr-xr-x. 1 xbitlabsin xbitlabsin 16824 Dec 18 11:27 ex2
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ./ex2
Error opening file!
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$
```

```
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ gcc example2.c -o ex2
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ls -lrt
total 52
-rw-r--r--. 1 xbitlabsin xbitlabsin   436 Dec 18 11:23 files.c
-rwxr-xr-x. 1 xbitlabsin xbitlabsin 16768 Dec 18 11:23 x
-rw-r--r--. 1 xbitlabsin xbitlabsin    43 Dec 18 11:23 test_file.txt
-rw-r--r--. 1 xbitlabsin xbitlabsin   442 Dec 18 11:26 example2.c
-rwxr-xr-x. 1 xbitlabsin xbitlabsin 16824 Dec 18 11:27 ex2
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ./ex2
Error opening file!
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ vim example2.c
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ vim example2.c
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ./ex2
Error opening file!
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ gcc example2.c -o ex2
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ./ex2
File Content:
Hello, World!
This is a file example in C.
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$
```

Alternatively, Using `fgetc` (Character-by-Character Reading):

```
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ vim 3ex.c
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ gcc 3ex.c -o 3ex
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ cat 3ex.c
#include <stdio.h>

int main() {
    FILE *file = fopen("test_file.txt", "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }

    int ch;
    while ((ch = fgetc(file)) != EOF) {
        putchar(ch);
    }

    fclose(file);
    return 0;
}

xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ./3ex
Hello, World!
This is a file example in C.
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$
```

Explanation

`fgetc` reads one character at a time from the file.

`putchar` prints each character to the console.

The loop stops when `fgetc` returns `EOF`.

```
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ls
3ex  3ex.c  ex2  example2.c  files.c  test_file.txt  x
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ vim 4exfile.c
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ cat 4exfile.c
#include <stdio.h>

int main() {
    FILE *file;

    // Open file for appending
    file = fopen("test_file.txt", "a");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // Append data to the file
    fprintf(file, "Adding another line to the file.\n");

    // Close the file
    fclose(file);

    printf("Data appended to file successfully.\n");
    return 0;
}

xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ gcc 4exfile.c -o 4ex
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ./4ex
Data appended to file successfully.
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ cat test_file.txt
Hello, World!
This is a file example in C.
Adding another line to the file.
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$
```

`fopen("example.txt", "a")`: Opens the file in append mode. Data is added to the end of the file without overwriting existing content.

`fprintf`: Appends a formatted string to the file.

Key Points

1. Always check if `fopen` returns `NULL` to handle errors (e.g., file not found, permission issues).
2. Use `fclose` after file operations to release resources.

3. Use appropriate modes (`r`, `w`, `a`, etc.) depending on the operation.
4. For binary data, use `fread` and `fwrite`

Binary File Operations

- `fwrite`: Writes raw binary data to a file.
- `fread`: Reads raw binary data from a file.
- Binary files are more efficient for storing large datasets but are not human-readable.

The function call:

```
fread(readNumbers, sizeof(int), 5, file);
```

will read 5 integers from the binary file (`binary.dat`) into the array `readNumbers`. If the file contains the data written in the earlier `fwrite` operation:

```
fwrite(numbers, sizeof(int), 5, file);
```

where `numbers[] = {10, 20, 30, 40, 50}`, the `fread` function will successfully copy these integers into `readNumbers`.

Steps:

1. File Contents:

- During `fwrite`, the file `binary.dat` was written with the binary representation of the integers `{10, 20, 30, 40, 50}`.
- Each integer occupies 4 bytes (on most systems where `sizeof(int) == 4`).

2. Reading the File:

- `fread` reads 5 blocks of size `sizeof(int)` (4 bytes each) from the file into the memory location pointed to by `readNumbers`.
- The array `readNumbers` will then contain the values `{10, 20, 30, 40, 50}`.

3. Output: If you print the contents of `readNumbers` as in the code:

```
for (int i = 0; i < 5; i++) {
    printf("%d ", readNumbers[i]);
}
```

The output will be:

```
10 20 30 40 50
```

```
Adding another line to the file.
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ vim 5_binary_file_op.c
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ gcc 5_binary_file_op.c -o 5bfo
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ cat 5_binary_file_op.c
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *file;
    int numbers[] = {10, 20, 30, 40, 50};
    int readNumbers[5];

    // Open file for writing binary data
    file = fopen("binary.dat", "wb");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // Write array to file
    fwrite(numbers, sizeof(int), 5, file);
    fclose(file);

    // Open file for reading binary data
    file = fopen("binary.dat", "rb");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // Read array from file
    fread(readNumbers, sizeof(int), 5, file);
    fclose(file);

    // Display read data
    printf("Read Numbers:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", readNumbers[i]);
    }
    printf("\n");

    return 0;
}

xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ ./5bfo
Read Numbers:
10 20 30 40 50
xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ cat binary.dat

(2xbitlabsin@fedora:~/Cavalier-December/sem1/files1$ cat binary.dat
```

Structures and Unions

A structure is a collection of variables (of different types) grouped together under one name. Each element in a structure is called a **member**.

Syntax:

```
struct StructName {
    data_type member1;
    data_type member2;
    // ... other members
};
```

main.c

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

// Define a structure
struct Employee {
    int id;
    char name[50];
    float salary;
};

int main() {
    // Declare a variable of type Employee
    struct Employee emp1;

    // Assign values to structure members
    emp1.id = 101;
    strcpy(emp1.name, "John Doe");
    emp1.salary = 75000.50;

    // Access and display structure members
    printf("Employee Details:\n");
    printf("ID: %d\n", emp1.id);
    printf("Name: %s\n", emp1.name);
    printf("Salary: %.2f\n", emp1.salary);

    return 0;
}
```

Output

```
Employee Details:
ID: 101
Name: John Doe
Salary: 75000.50


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

- Members are stored in **different memory locations**.
- Each member has its own data type and size.
- **Nested Structures**: Structures can be nested.
- Supports **arrays of structures**.

Unions

A union is similar to a structure but uses a **shared memory location** for all its members. Only **one member can store a value at a time**.

Syntax:

```
union UnionName {

    data_type member1;

    data_type member2;

    // ... other members
```

```
};
```

main.c

```c
#include <stdio.h>

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

int main() {
    // Declare a variable of type Data
    union Data data;

    // Assign and display values for each member
    data.intValue = 10;
    printf("Integer: %d\n", data.intValue);

    data.floatValue = 5.5;
    printf("Float: %.2f\n", data.floatValue);

    data.charValue = 'A';
    printf("Character: %c\n", data.charValue);

    // Note: Only the last assigned member is valid
    printf("Integer (after assigning char): %d\n", data.intValue); // Undefined behavior

    return 0;
}
```

Output

```
Integer: 10
Float: 5.50
Character: A
Integer (after assigning char): 1085276225


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

- All members share the **same memory location**.
- **Size of the union** is equal to the size of its largest member.
- Only **one member can hold a value at a time**.

Differences Between Structures and Unions

Feature	Structure	Union
Memory Allocation	Allocates separate memory for each member.	Shares memory among all members.
Size	Sum of the sizes of all members.	Size of the largest member.
Access	All members can be accessed simultaneously.	Only one member holds a valid value at a time.
Use Case	Used when storing related data that can coexist.	Used when variables store mutually exclusive data.

---

main.c

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

// Define a structure and a union
struct Student {
    int id;
    char name[20];
    float marks;
};

union Employee {
    int id;
    char name[20];
    float salary;
};

int main() {
    struct Student student = {1, "Alice", 85.5};
    union Employee employee;

    // Assign values to the union
    employee.id = 1001;
    printf("Union (ID): %d\n", employee.id);

    strcpy(employee.name, "Bob");
    printf("Union (Name): %s\n", employee.name);

    employee.salary = 75000.50;
    printf("Union (Salary): %.2f\n", employee.salary);

    // Structure example
    printf("\nStructure (ID): %d\n", student.id);
    printf("Structure (Name): %s\n", student.name);
    printf("Structure (Marks): %.2f\n", student.marks)

    return 0;
}
```

Output

```
Union (ID): 1001
Union (Name): Bob
Union (Salary): 75000.50

Structure (ID): 1
Structure (Name): Alice
Structure (Marks): 85.50


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

1. **Structures**: Use when members need to hold independent data simultaneously.
    - Example: Storing data about a student or an employee.
2. **Unions**: Use when only one member holds valid data at a time.
    - Example: Representing a value that can be of multiple types (e.g., sensor data).

Both structures and unions provide a flexible way to manage data, with their choice depending on the memory and access requirements of your application.

Convert a string to uppercase

```c
#include <stdio.h>

int main() {
```

```
    char str[100]; // vijeta\0
    printf("Enter a string:\n");
    gets(str);

    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] >= 'a' && str[i] <= 'z') {
            str[i] = str[i] - 32;
        }
    }

    printf("Uppercase string: %s\n", str);
    return 0;
}
```

`str[i] >= 'a'`: Checks if the character is greater than or equal to `'a'` (ASCII value 97).

`str[i] <= 'z'`: Checks if the character is less than or equal to `'z'` (ASCII value 122).

In the ASCII table, the difference between the lowercase letter and its corresponding uppercase letter is `32`. For example:

- `'a'` (ASCII 97) - `32` = `'A'` (ASCII 65)
- `'b'` (ASCII 98) - `32` = `'B'` (ASCII 66)

END