Dr. SNS RAJALAKSHMI COLLEGE OF ARTS AND SCIENCE

(AUTONOMOUS)

Accredited by NAAC (Cycle- III) with 'A+' Grade

DEPARTMENT OF B.SC CS (GCD)

23UCU401 – PROGRAMMING IN C
UNIT- V

Subject : Programming in C
Department : GCD
Faculty : Mrs.K.Sangeetha

Unit 5 :

Why Use Structures – Array of Structures - Additional Features of Structures Uses of Structures. File Input/Output: File Operations - Closing the File - Counting Characters, Tabs, Spaces - A File-copy Program - File Opening Modes. Miscellaneous Features: Enumerated Data Type - Renaming Data Types with typedef -Typecasting - Bit Fields – Unions.

1. Why Use Structures

- Structures allow you to group related variables into a single entity.
- They enhance code organization and readability.

Example:

```
struct Point {
    int x;
    int y;
};

struct Point p;
p.x = 10;
p.y = 5;
```

Explanation:

- The structure `Point` groups `x` and `y` coordinates together.
- Creating a variable `p` of this structure type allows us to access both coordinates easily.

2. Array of Structures

- You can create arrays of structures to manage multiple records of the same structure type.
- This is useful for tasks like managing data for multiple students.

Example:

```
struct Student {
    char name[50];
    int age;
};

struct Student students[3];
students[0].age = 20;
students[1].age = 22;
students[2].age = 19;
```

Explanation:

- The array `students` contains structures to store information about multiple students.
- Each student's age is assigned to the `age` field.

3. Additional Features of Structures Uses of Structures

- Structures can be nested (a structure within a structure) for representing complex data hierarchies.
- They are used extensively in applications like databases, file formats, and data exchange between programs.

```
struct Address {
    char street[50];
    char city[30];
    char state[20];
};

struct Person {
    char name[50];
    struct Address address;
};
```

Explanation:

- The `Person` structure is nested within the `Address` structure, allowing you to represent a person's name and address in a hierarchical manner.

4. File Input/Output: File Operations

- File operations in C are essential for reading from and writing to external files.
- The basic file operations include opening, reading, writing, and closing files.

Example: Reading from a File

```
FILE *file = fopen("data.txt", "r");
char buffer[100];
if (file != NULL) {
    while (fgets(buffer, sizeof(buffer), file) != NULL) {
        printf("%s", buffer);
    }
    fclose(file);
}
```

Explanation:

- The program opens a file named "data.txt" in read mode, reads its content line by line, and prints it.

5. Closing the File

- Properly closing files is essential to release system resources and ensure data integrity.
- Use `fclose()` to close a file after reading or writing.

6. Counting Characters

- When reading characters from a file, you may need to count the number of characters, words, or lines.
- This is useful for text analysis and processing.

Example: Counting Characters in a File

```
FILE *file = fopen("data.txt", "r");
int count = 0;
if (file != NULL) {
    int ch;
    while ((ch = fgetc(file)) != EOF) {
        if (ch != ' ' && ch != '\n' && ch != '\t') {
            count++;
        }
    }
    fclose(file);
}
```

Explanation:

- The program counts the number of characters in a file while ignoring spaces, tabs, and line breaks.

7. Tabs, Spaces

- When processing text files, you may encounter tabs, spaces, and newline characters.
- You can write programs to count, replace, or manipulate these characters in text data.

Example: Replacing Tabs with Spaces

```
FILE *inputFile = fopen("input.txt", "r");
FILE *outputFile = fopen("output.txt", "w");

if (inputFile != NULL && outputFile != NULL) {
    int ch;
    while ((ch = fgetc(inputFile)) != EOF) {
        if (ch == '\t') {
            fprintf(outputFile, "    "); // Replace tab with 4 spaces
        } else {
            fputc(ch, outputFile);
        }
    }
    fclose(inputFile);
    fclose(outputFile);
}
```

Explanation:

- The program reads a file, replaces tabs with spaces, and writes the modified content to another file.

8. A File-copy Program

- Writing a file-copy program is a common exercise in file operations.
- It involves opening a source file, reading its content, and writing it to a destination file.

Example: File Copy Program

```
FILE *source = fopen("source.txt", "r");
FILE *destination = fopen("destination.txt", "w");

if (source != NULL && destination != NULL) {
    int ch;
    while ((ch = fgetc(source)) != EOF) {
        fputc(ch, destination);
    }
    fclose(source);
    fclose(destination
```

Explanation:

- The program reads from a source file and writes its content to a destination file.

9. File Opening Modes

- File opening modes determine how a file can be accessed and modified.
- Understanding file modes is crucial to control file access and prevent data loss.
- Common file modes include:
  - `"r"`: Read
  - `"w"`: Write
  - `"a"`: Append
  - `"rb"`: Read as binary
  - `"wb"`: Write as binary

10. Miscellaneous Features: Enumerated Data Type -

Enumerated data types allow you to define a set of named integer constants. - They make code more readable and self-documenting when dealing with specific values.

Example: Defining an Enumeration

```c
enum Weekdays {
    MON, TUE, WED, THU, FRI
};

enum Weekdays today = WED;
```

Explanation:
- The `enum` defines named constants for weekdays.
- We assign the value `WED` to the variable `today`.

11. Renaming Data Types with typedef -

The `typedef` keyword is used to create new data type names. - It makes the code more expressive and can help abstract data types for better portability.

Example: Typedef for Data Type

```c
typedef unsigned int uint; // Define 'uint' as an alias for 'unsigned int'
uint x = 42;
```

Explanation:
- The `typedef` statement creates an alias `uint` for the data type `unsigned int`.
- This improves code readability and portability.

12. Typecasting

Typecasting is the conversion of one data type to another. - It's useful when you need to perform operations with data of different types.

Example: Typecasting

```c
double pi = 3.14159;
int piInt = (int)pi; // Typecast 'pi' to an integer
```

Explanation:
- Typecasting `pi` to an integer truncates the decimal part and assigns it to `piInt`.

13. Bit Fields

Bit fields are used to control the size of data fields within a structure. - They are often used in embedded systems programming and data compression.

Example: Bit Fields

```c
struct Flags {
    unsigned int flag1 : 1;
    unsigned int flag2 : 1;
    unsigned int flag3 : 1;
};

struct Flags status;
status.flag1 = 1;
```

Explanation:
- The structure `Flags` uses bit fields to represent individual flags with a size of 1 bit each.
- This is space-efficient for storing binary flags.

14. Unions

Unions are similar to structures but share the same memory location for their members. - They are useful when you want to represent data that can be one of several types at a given time.

Example: Union for Multiple Data Types

```c
union Data {
    int integer;
    double floating;
};

union Data value;
value.integer = 42;
```

Explanation:
- The `Data` union can store either an integer or a floating-point value.
- This is useful when you need to handle different types of data in the same memory space.