File Handling In C

Option 1: Copy-Paste into Microsoft Word (or another word processor)

1. **Copy the text below:**
   (You can copy everything from the content below.)
2. **Paste into a new document** in your favorite word processor (e.g., Microsoft Word, LibreOffice Writer).
3. **Format as needed** (adjust fonts, headings, etc.) and then save or export as PDF.

Option 2: Use Markdown and Convert with Pandoc

1. **Copy the Markdown content below** into a file (e.g., `tutorial.md`).
2. **Convert using Pandoc** with a command like:
   `bash    pandoc tutorial.md -o tutorial.pdf`
   Or convert it to a Word document:
   `bash    pandoc tutorial.md -o tutorial.docx`

Below is the complete tutorial in Markdown format:

```markdown # Advanced File Handling and Management in C

This tutorial covers advanced file handling and management in C. It is designed for intermediate to advanced developers and includes both theoretical concepts and sample code for the following topics:

1. **File Handling and Management**
   1.1. File Operations
   - File Descriptors and Streams
   - Opening and Closing Files
   - Reading and Writing Data
   - File Positioning
   1.2. File Attributes and Permissions
   - Retrieving File Information
   - Modifying File Permissions and Ownership
   - Handling File Metadata
   1.3. Directory Handling
   - Working with Directories
   - Listing Directory Contents
   - Navigating the File System

---
```

1. File Handling and Management in C

File I/O in C can be approached using two main interfaces:

- **Low-level (POSIX) System Calls:**
  These use integer file descriptors (e.g., `open()`, `read()`, `write()`, `lseek()`, `close()`) and provide granular control.

- **High-level Standard I/O Library:**
  These use `FILE *` streams (e.g., `fopen()`, `fread()`, `fwrite()`, `fseek()`, `fclose()`) and offer buffered I/O with convenience functions.

Understanding both approaches is essential for robust and efficient C programming.

1.1. File Operations

1.1.1. File Descriptors and Streams

Theory:
- **File Descriptors:**
 Low-level handles represented by an integer. They are used with `open()`, `read()`, `write()`, and `lseek()`.

- **Streams:**
  Higher-level abstractions provided by the C Standard Library. They are used with functions like `fgetc()`, `fprintf()`, and `fread()`, and obtained with `fopen()`.

Sample Code: Converting a File Descriptor to a Stream

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    // Open a file using the POSIX system call (returns a file descriptor)
    int fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    // Convert the file descriptor to a standard I/O stream
    FILE *fp = fdopen(fd, "r");
    if (!fp) {
        perror("fdopen");
        close(fd);
        return 1;
    }

    // Use the stream to read data (for example, using fgetc)
    int ch = fgetc(fp);
    if (ch != EOF)
```

```
        putchar(ch);

    // Close the stream (this also closes the underlying file descriptor)
    fclose(fp);
    return 0;
}
```

1.1.2. Opening and Closing Files

Theory:
- **Standard I/O:** Use `fopen()` and `fclose()`.
- **POSIX System Calls:** Use `open()` and `close()`.
- Always check for errors after opening a file.

Sample Code: Using Standard I/O Functions

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

int main(void) {
    // Open a file for reading and writing ("w+" creates or truncates the
file)
    FILE *fp = fopen("data.txt", "w+");
    if (fp == NULL) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    // Perform file operations here...

    // Always close the file when done
    fclose(fp);
    return 0;
}
```

1.1.3. Reading and Writing Data

Theory:
- **Writing:** Use functions like `fprintf()`, `fwrite()`, or low-level `write()`. - **Reading:** Use `fgets()`, `fread()`, or low-level `read()`. - Be aware of buffering when using streams; `fflush()` can be used to force data writing.

Sample Code: Writing and Then Reading Using Standard I/O

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

int main(void) {
    FILE *fp = fopen("data.txt", "w+");
    if (fp == NULL) {
```

```c
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    // Write data to the file using fprintf
    fprintf(fp, "Hello, world!\n");

    // Flush to ensure data is written to disk
    fflush(fp);

    // Rewind the file pointer to the beginning for reading
    fseek(fp, 0, SEEK_SET);

    // Read data from the file using fgets
    char buffer[128];
    if (fgets(buffer, sizeof(buffer), fp) != NULL) {
        printf("Read: %s", buffer);
    } else {
        perror("fgets");
    }

    fclose(fp);
    return 0;
}
```

Alternate: Using POSIX System Calls

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

int main(void) {
    // Open the file for reading and writing (create if not exists)
    int fd = open("data.txt", O_RDWR | O_CREAT, 0644);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    const char *text = "Hello, POSIX world!\n";
    // Write data to the file
    if (write(fd, text, strlen(text)) == -1) {
        perror("write");
        close(fd);
        return 1;
    }

    // Position file pointer to the beginning
    if (lseek(fd, 0, SEEK_SET) == -1) {
        perror("lseek");
        close(fd);
        return 1;
    }

    char buf[128];
    ssize_t bytesRead = read(fd, buf, sizeof(buf) - 1);
    if (bytesRead == -1) {
        perror("read");
        close(fd);
        return 1;
    }
    buf[bytesRead] = '\0';
    printf("Read: %s", buf);

    close(fd);
    return 0;
}
```

1.1.4. File Positioning

Theory:
- **Standard I/O:** Use `fseek()`, `ftell()`, and `rewind()` to navigate within a file. - **POSIX:** Use `lseek()` to adjust the file offset.

Sample Code: Using fseek, ftell, and rewind

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

int main(void) {
    FILE *fp = fopen("data.txt", "r+");
    if (!fp) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    // Seek to the end of the file
    if (fseek(fp, 0, SEEK_END) != 0) {
        perror("fseek");
        fclose(fp);
        exit(EXIT_FAILURE);
    }

    // Get the current position (i.e., file size)
    long size = ftell(fp);
    if (size == -1L) {
        perror("ftell");
        fclose(fp);
        exit(EXIT_FAILURE);
    }
    printf("File size: %ld bytes\n", size);

    // Reset file pointer to the beginning of the file
    rewind(fp);
```

```c
    fclose(fp);
    return 0;
}
```

Sample Code: Using lseek with File Descriptors

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    int fd = open("data.txt", O_RDWR);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    // Move file pointer to end and retrieve its offset
    off_t offset = lseek(fd, 0, SEEK_END);
    if (offset == (off_t)-1) {
        perror("lseek");
        close(fd);
        return 1;
    }
    printf("File size: %ld bytes\n", (long)offset);

    close(fd);
    return 0;
}
```

1.2. File Attributes and Permissions

Managing file attributes is crucial for security, ownership, and metadata.

1.2.1. Retrieving File Information

Theory:
- Use the `stat()` system call to populate a `struct stat` with file information (size, permissions, timestamps, etc.).

Sample Code: Using stat()

```c
#include <stdio.h>
#include <sys/stat.h>
#include <time.h>

int main(void) {
    struct stat sb;
    if (stat("data.txt", &sb) == -1) {
        perror("stat");
        return 1;
    }

    printf("File size: %ld bytes\n", (long)sb.st_size);
    printf("Last modified: %s", ctime(&sb.st_mtime));

    return 0;
}
```

1.2.2. Modifying File Permissions and Ownership

Theory:
- **chmod():** Change file permissions. - **chown():** Change file owner and group (requires appropriate privileges). - **umask:** Set default permission masks for new files.

Sample Code: Changing File Permissions with chmod()

```
#include <stdio.h>
#include <sys/stat.h>
#include <stdlib.h>

int main(void) {
    // Set permissions to rwxr-xr-- (owner: read, write, execute; group: 
read, execute; others: read)
    if (chmod("data.txt", S_IRUSR | S_IWUSR | S_IXUSR |
                         S_IRGRP | S_IXGRP |
                         S_IROTH) == -1) {
        perror("chmod");
        exit(EXIT_FAILURE);
    }
    printf("Permissions changed successfully.\n");
    return 0;
}
```

Sample Code: Changing File Ownership with chown()

Note: Changing file ownership typically requires superuser privileges.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main(void) {
    // Change file owner to UID 1001 and group to GID 1001
    if (chown("data.txt", 1001, 1001) == -1) {
        perror("chown");
        exit(EXIT_FAILURE);
    }
    printf("Ownership changed successfully.\n");
    return 0;
}
```

1.2.3. Handling File Metadata

Theory:
- File metadata (timestamps, extended attributes, etc.) can be managed with various system calls. - `utime()` or `utimes()` can update access and modification times. - Extended attributes can be manipulated using `getxattr()`/`setxattr()` on supported systems.

Sample Code: Updating File Timestamps with utime()

```c
#include <stdio.h>
#include <sys/stat.h>
#include <utime.h>
#include <time.h>

int main(void) {
    struct utimbuf new_times;
    // Set both access and modification times to the current time
    new_times.actime = time(NULL);
    new_times.modtime = new_times.actime;

    if (utime("data.txt", &new_times) == -1) {
        perror("utime");
        return 1;
    }
    printf("File timestamps updated successfully.\n");
    return 0;
}
```

1.3. Directory Handling

Directories are special files that contain entries for other files and subdirectories.

1.3.1. Working with Directories

Theory:
- Use `opendir()` to open a directory. - Use `readdir()` to read directory entries. - Use `closedir()` to close the directory stream.

Sample Code: Reading a Directory

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

int main(void) {
    DIR *dir = opendir(".");
    if (!dir) {
        perror("opendir");
        exit(EXIT_FAILURE);
    }
    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }

    closedir(dir);
    return 0;
}
```

1.3.2. Listing Directory Contents with File Type Distinction

Theory:
- After reading each directory entry, use `stat()` (or `lstat()` for symbolic links) to determine the entry type (file, directory, etc.). - This helps in filtering and organizing the output.

Sample Code: Listing Files and Directories

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>
#include <string.h>

int main(void) {
    DIR *dir = opendir(".");
    if (!dir) {
        perror("opendir");
        exit(EXIT_FAILURE);
    }

    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        // Skip current and parent directory entries
        if (strcmp(entry->d_name, ".") == 0 ||
            strcmp(entry->d_name, "..") == 0)
            continue;

        struct stat sb;
        if (stat(entry->d_name, &sb) == 0) {
            if (S_ISDIR(sb.st_mode))
                printf("[DIR]  %s\n", entry->d_name);
            else
                printf("[FILE] %s\n", entry->d_name);
        } else {
            perror("stat");
        }
    }

    closedir(dir);
    return 0;
}
```

1.3.3. Navigating the File System

Theory:
- **chdir():** Changes the current working directory. - **getcwd():** Retrieves the current working directory. - These functions allow dynamic navigation within the file system.

Sample Code: Changing and Getting the Current Directory

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

int main(void) {
    // Change the current working directory to /tmp
    if (chdir("/tmp") == -1) {
        perror("chdir");
        exit(EXIT_FAILURE);
    }

    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("Current working directory: %s\n", cwd);
    } else {
        perror("getcwd");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Summary

In this tutorial, we covered:

- **File Operations:**
  Differentiating between file descriptors and streams; opening/closing files; reading/writing; and file positioning.
- **File Attributes and Permissions:**
  Retrieving metadata using `stat()`, modifying permissions with `chmod()` and ownership with `chown()`, and updating timestamps with `utime()`.
- **Directory Handling:**
  Opening directories with `opendir()`, reading their contents with `readdir()`, distinguishing file types via `stat()`, and navigating the file system using `chdir()` and `getcwd()`.