8.a. Write a C program to read n characters from a file and append them back to the same file using dup2 function.

Here's a C program that reads n characters from a file and appends them back to the same file using the dup2 function:

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

#define MAX_SIZE 100

int main() {
    char filename[] = "file.txt";
    int fd, new_fd;
    char buffer[MAX_SIZE];
    ssize_t bytes_read;
    int n;

    printf("Enter the number of characters to read and append: ");
    scanf("%d", &n);

    // Open the file
    fd = open(filename, O_RDWR);
    if (fd == -1) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    // Move file pointer to the end
    lseek(fd, 0, SEEK_END);

    // Duplicate file descriptor
    new_fd = dup(fd);
    if (new_fd == -1) {
        perror("Error duplicating file descriptor");
        close(fd);
        exit(EXIT_FAILURE);
    }

    // Move the file pointer back 'n' characters
    lseek(fd, -n, SEEK_END);

    // Read 'n' characters from the file
    bytes_read = read(fd, buffer, n);
    if (bytes_read == -1) {
        perror("Error reading file");
```

```
        close(fd);
        close(new_fd);
        exit(EXIT_FAILURE);
    }

    // Append the read characters back to the file
    if (write(new_fd, buffer, bytes_read) != bytes_read) {
        perror("Error writing to file");
        close(fd);
        close(new_fd);
        exit(EXIT_FAILURE);
    }

    // Close file descriptors
    close(fd);
    close(new_fd);

    printf("%d characters read and appended successfully.\n", (int)bytes_read);

    return 0;
}
```

Explanation:

1. We include necessary header files like **<stdio.h>, <stdlib.h>, <unistd.h>, and <fcntl.h>** for file operations and standard I/O functions.
2. We define a macro **MAX_SIZE** for the maximum buffer size.
3. In the **main()** function:
    - We prompt the user to enter the number of characters to read and append.
    - Open the file in read-write mode **(O_RDWR)**.
    - Move the file pointer to the end of the file using **lseek()** to prepare for appending.
    - Duplicate the file descriptor using **dup()** to have two descriptors pointing to the same file.
    - Move the file pointer back 'n' characters from the end using **lseek()**.
    - Read 'n' characters from the file using **read()**.
    - Append the read characters back to the file using **write()**.
    - Close the file descriptors.

Execution Steps:

1. Compile the program using a C compiler (e.g., **gcc -o read_and_append read_and_append.c**).
2. Create a text file named **file.txt** with some content.
3. Run the compiled program (**./read_and_append**) .
4. Enter the number of characters to read and append.
5. The program reads the specified number of characters from the end of the file and appends them back to the same file.
6. Verify the changes in **file.txt**.

8.b. Write a C program to list all files in a current directory using system calls.

To list all files in the current directory using system calls in C, you can use the **opendir**, **readdir**, and **closedir** functions. Here's a simple C program to achieve this:

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

int main() {
   DIR *dir;
   struct dirent *entry;

   // Open the current directory
   dir = opendir(".");
   if (dir == NULL) {
      perror("Error opening directory");
      exit(EXIT_FAILURE);
   }

   // Read and print all files in the directory
   while ((entry = readdir(dir)) != NULL) {
      if (entry->d_type == DT_REG) {
         printf("%s\n", entry->d_name);
      }
   }

   // Close the directory
   closedir(dir);

   return 0;
}
```

Explanation:

1. We include necessary header files like **<stdio.h>, <stdlib.h>, and <dirent.h>** for directory-related operations.
2. In the **main()** function:
   - We declare a **DIR** pointer **(dir)** to represent the directory stream and a **struct dirent** pointer **(entry)** to represent a directory entry.
   - Open the current directory (.) using **opendir()** and handle errors if the directory cannot be opened.
   - Loop through each entry in the directory using **readdir()**. Print the names of regular files (**DT_REG**).
   - Close the directory using **closedir()**.

Execution Steps:

1. Compile the program using a C compiler (e.g., **gcc -o list_files list_files.c**).
2. Navigate to the directory you want to list files in using the terminal.
3. Run the compiled program (**./list_files**).
4. The program will print the names of all regular files in the current directory.

Note: This program lists only regular files (**DT_REG**). If you want to list directories as well, you can modify the condition inside the **while** loop to include directories (**DT_DIR**).

---

9. a. Create a C program to simulate the copy command in Unix (cp command).

Here's a simple C program to simulate the copy command in Unix (similar to `cp` command). This program copies the content of one file to another:

Code:

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

#define BUFFER_SIZE 1024

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source_file> <destination_file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    FILE *source_file, *destination_file;
    char buffer[BUFFER_SIZE];
    size_t bytesRead;

    // Open the source file in binary read mode
    source_file = fopen(argv[1], "rb");
    if (source_file == NULL) {
        perror("Error opening source file");
        exit(EXIT_FAILURE);
    }

    // Open or create the destination file in binary write mode
    destination_file = fopen(argv[2], "wb");
    if (destination_file == NULL) {
        perror("Error opening destination file");
        fclose(source_file);
        exit(EXIT_FAILURE);
    }

    // Copy the content from source to destination
    while ((bytesRead = fread(buffer, 1, BUFFER_SIZE, source_file)) > 0) {
        fwrite(buffer, 1, bytesRead, destination_file);
    }

    // Close the files
    fclose(source_file);
    fclose(destination_file);
```

```
    printf("File copied successfully.\n");

    return 0;
}
```

Explanation:

1. We include necessary header files like **<stdio.h>** and **<stdlib.h>**.
2. In the **main()** function:
    - Check if the program is called with exactly two arguments (source file and destination file).
    - Open the source file in binary read mode (**"rb"**).
    - Open or create the destination file in binary write mode (**"wb"**).
    - Use a buffer to read from the source file and write to the destination file.
    - Copy the content of the source file to the destination file in chunks.
    - Close both the source and destination files.

Execution Steps:

1. Compile the program using a C compiler (e.g., **gcc -o my_cp my_cp.c**).
2. Run the compiled program with the source and destination file names as arguments (e.g., **./my_cp source.txt destination.txt**).
3. Check the destination file to verify that the content has been copied.

9. b. Develop a C program to simulate the ls (list) Unix command using system calls.

To simulate the `ls` (list) Unix command using system calls in C, you can use the `opendir`, `readdir`, and `closedir` functions. Here's a simple C program to achieve this:

Code:

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

int main() {
    DIR *dir;
    struct dirent *entry;

    // Open the current directory
    dir = opendir(".");
    if (dir == NULL) {
        perror("Error opening directory");
        exit(EXIT_FAILURE);
    }

    // Read and print all files and directories in the directory
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }

    // Close the directory
    closedir(dir);

    return 0;
}
```

Explanation:

1. We include necessary header files like **<stdio.h>**, **<stdlib.h>**, and **<dirent.h>** for directory-related operations.
2. In the **main()** function:
    - We declare a **DIR** pointer (**dir**) to represent the directory stream and a **struct dirent** pointer (**entry**) to represent a directory entry.
    - Open the current directory (**.**) using **opendir()** and handle errors if the directory cannot be opened.

- Loop through each entry in the directory using **readdir()** and print the names of all files and directories.
- Close the directory using **closedir()**.

Execution Steps:

1. Compile the program using a C compiler (e.g., **gcc -o my_ls my_ls.c**).
2. Run the compiled program (**./my_ls**).
3. The program will print the names of all files and directories in the current directory.

Note: This program lists both files and directories. If you want to list only files or only directories, you can modify the condition inside the **while** loop accordingly (**DT_REG** for files and **DT_DIR** for directories).