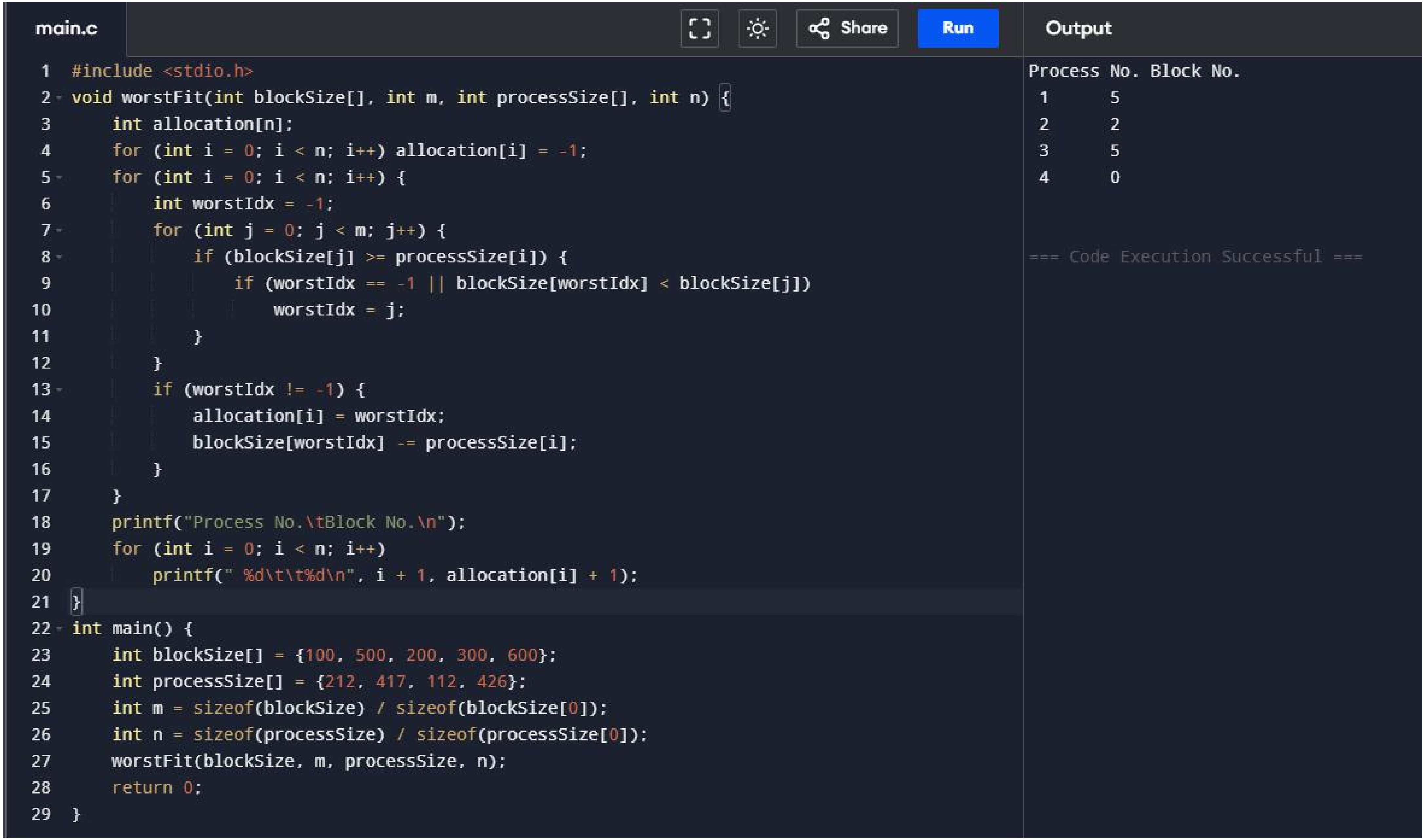
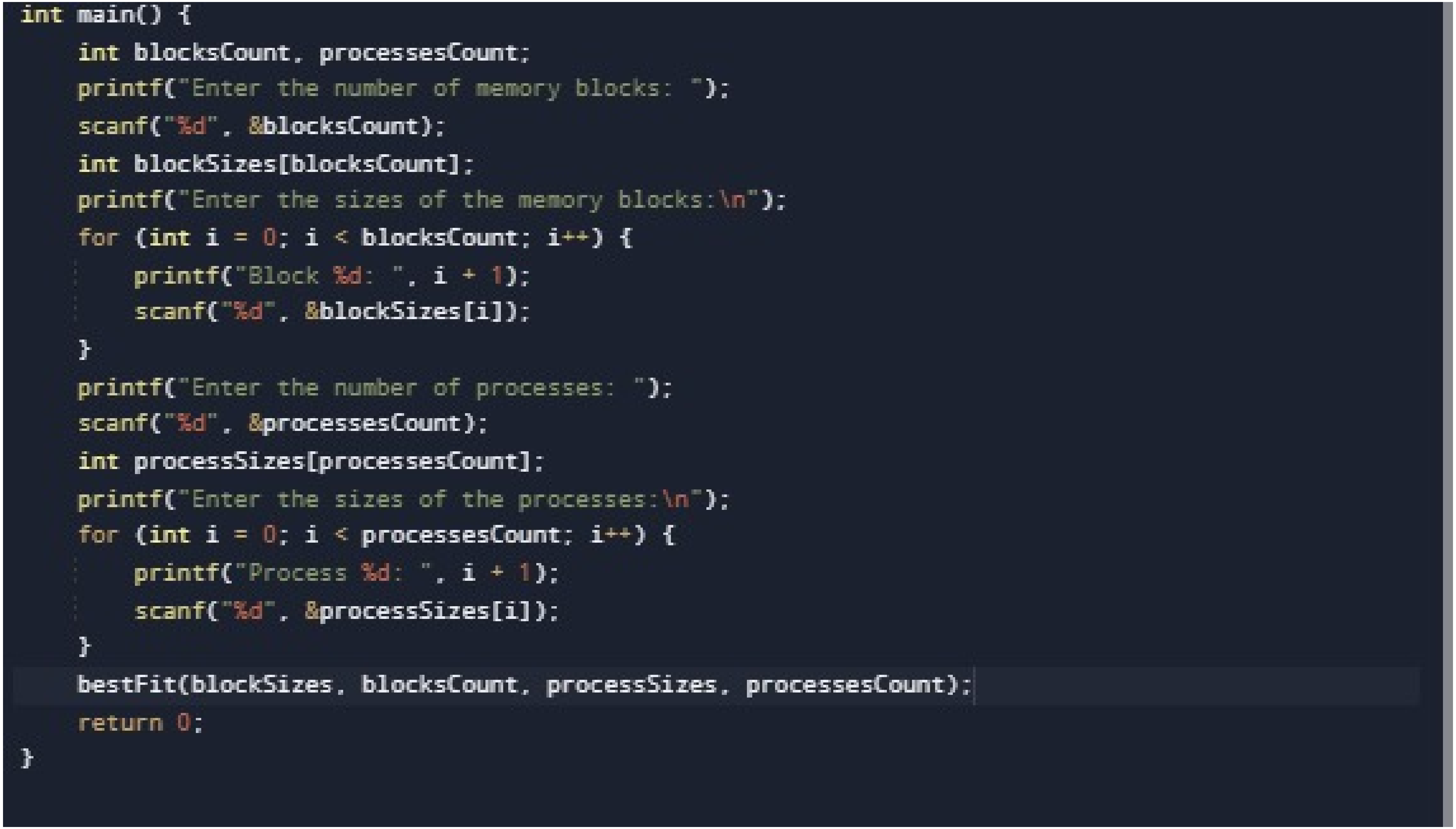
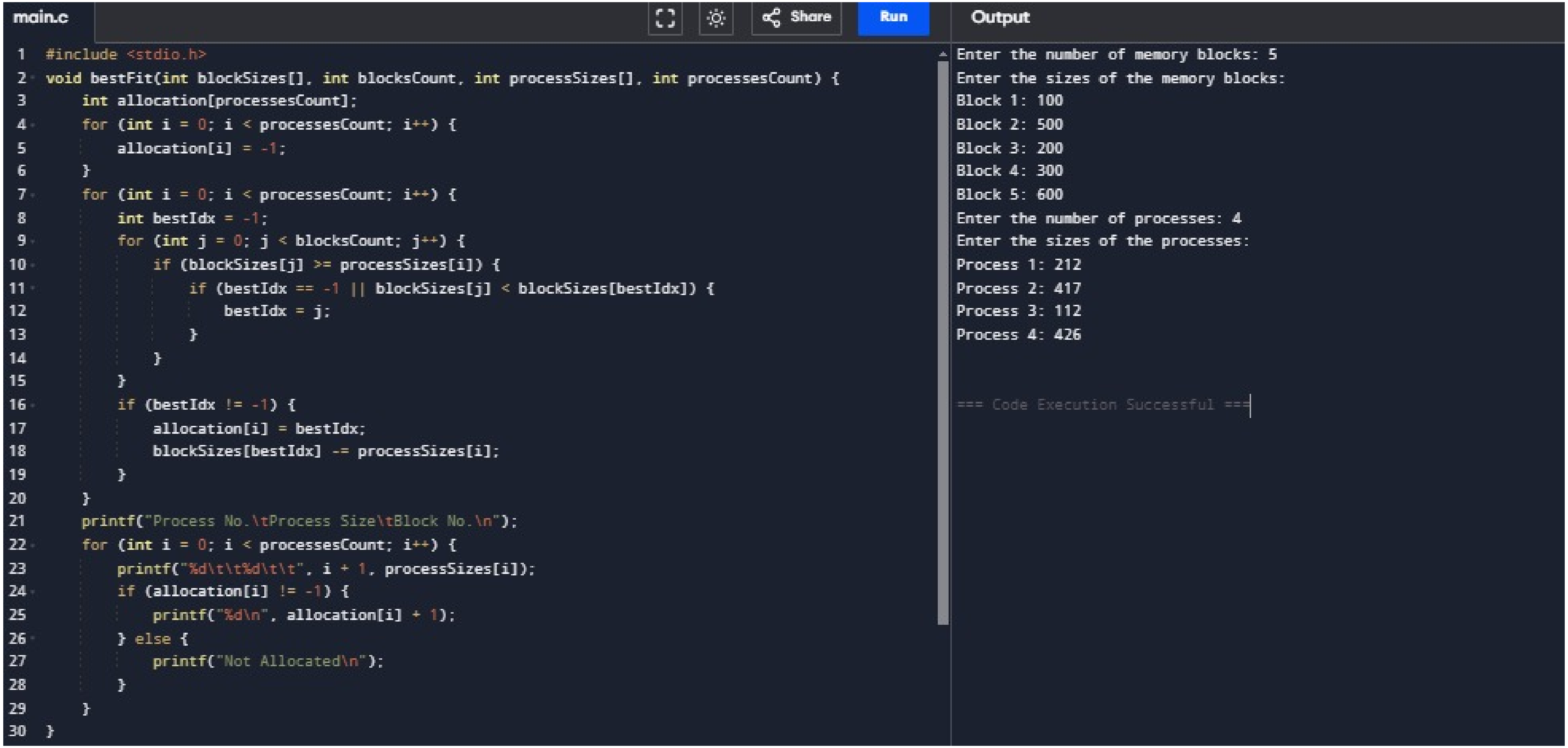


```c
#include <stdio.h>
void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) allocation[i] = -1;
    for (int i = 0; i < n; i++) {
        int worstIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j])
                    worstIdx = j;
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blockSize[worstIdx] -= processSize[i];
        }
    }
    printf("Process No.\tBlock No.\n");
    for (int i = 0; i < n; i++)
        printf(" %d\t\t%d\n", i + 1, allocation[i] + 1);
}
int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);
    worstFit(blockSize, m, processSize, n);
    return 0;
}
```

```
Process No. Block No.
 1      5
 2      2
 3      5
 4      0


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

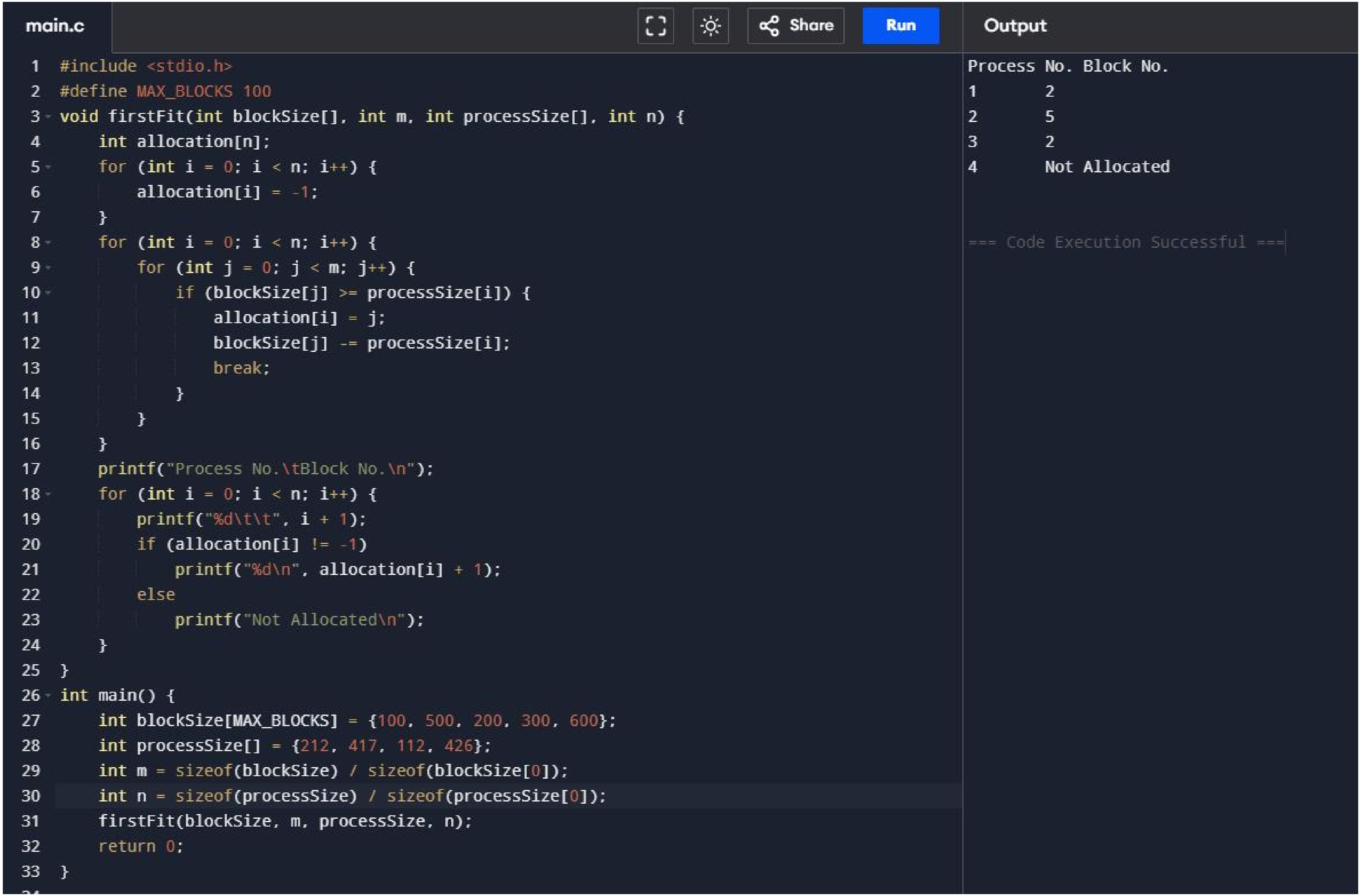


```
int main() {
    int blocksCount, processesCount;
    printf("Enter the number of memory blocks: ");
    scanf("%d", &blocksCount);
    int blockSizes[blocksCount];
    printf("Enter the sizes of the memory blocks:\n");
    for (int i = 0; i < blocksCount; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSizes[i]);
    }
    printf("Enter the number of processes: ");
    scanf("%d", &processesCount);
    int processSizes[processesCount];
    printf("Enter the sizes of the processes:\n");
    for (int i = 0; i < processesCount; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &processSizes[i]);
    }
    bestFit(blockSizes, blocksCount, processSizes, processesCount);
    return 0;
}
```



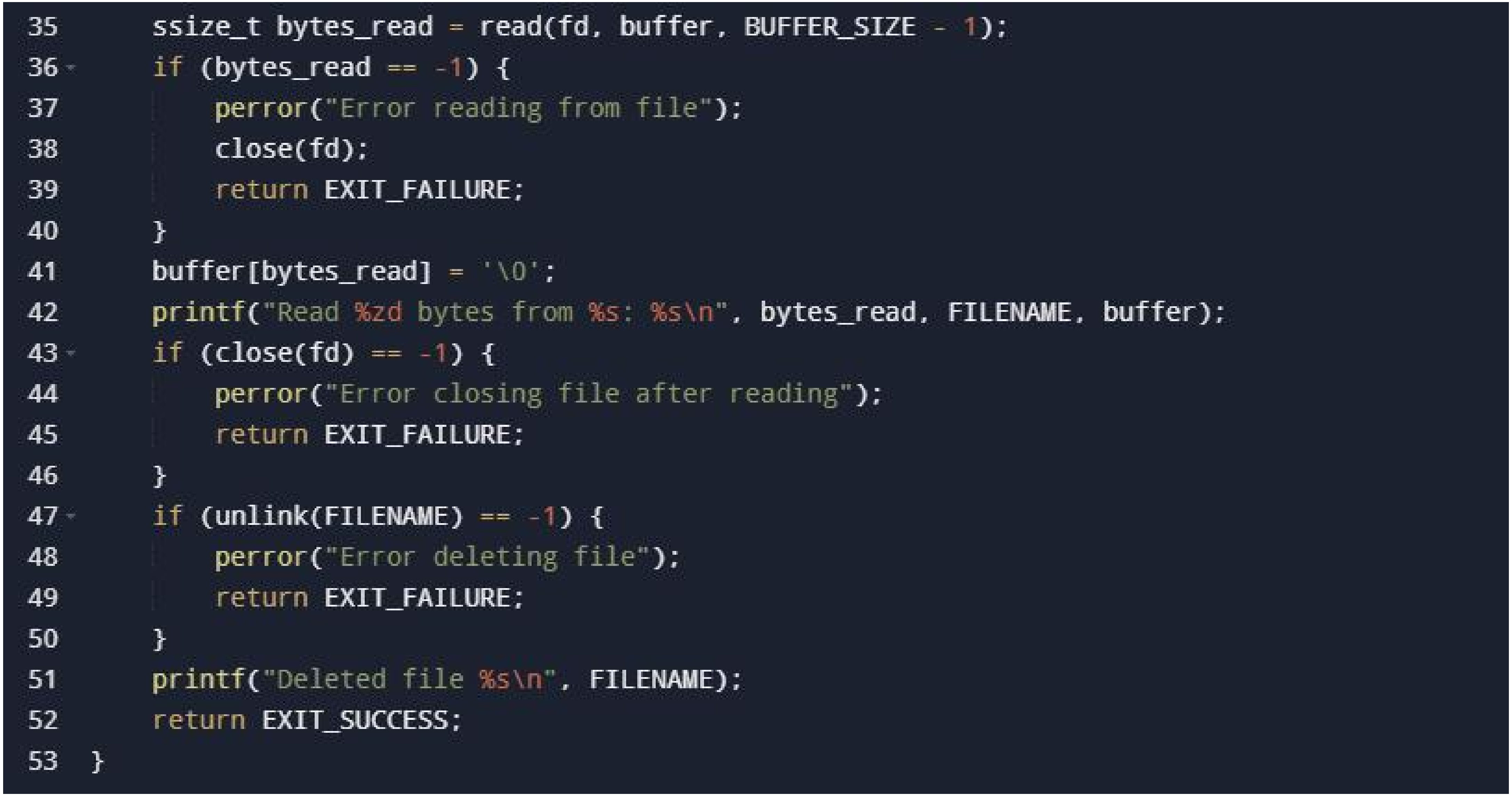
```c
#include <stdio.h>
void bestFit(int blockSizes[], int blocksCount, int processSizes[], int processesCount) {
    int allocation[processesCount];
    for (int i = 0; i < processesCount; i++) {
        allocation[i] = -1;
    }
    for (int i = 0; i < processesCount; i++) {
        int bestIdx = -1;
        for (int j = 0; j < blocksCount; j++) {
            if (blockSizes[j] >= processSizes[i]) {
                if (bestIdx == -1 || blockSizes[j] < blockSizes[bestIdx]) {
                    bestIdx = j;
                }
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blockSizes[bestIdx] -= processSizes[i];
        }
    }
    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < processesCount; i++) {
        printf("%d\t\t%d\t\t", i + 1, processSizes[i]);
        if (allocation[i] != -1) {
            printf("%d\n", allocation[i] + 1);
        } else {
            printf("Not Allocated\n");
        }
    }
}
```

```
Enter the number of memory blocks: 5
Enter the sizes of the memory blocks:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600
Enter the number of processes: 4
Enter the sizes of the processes:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426


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

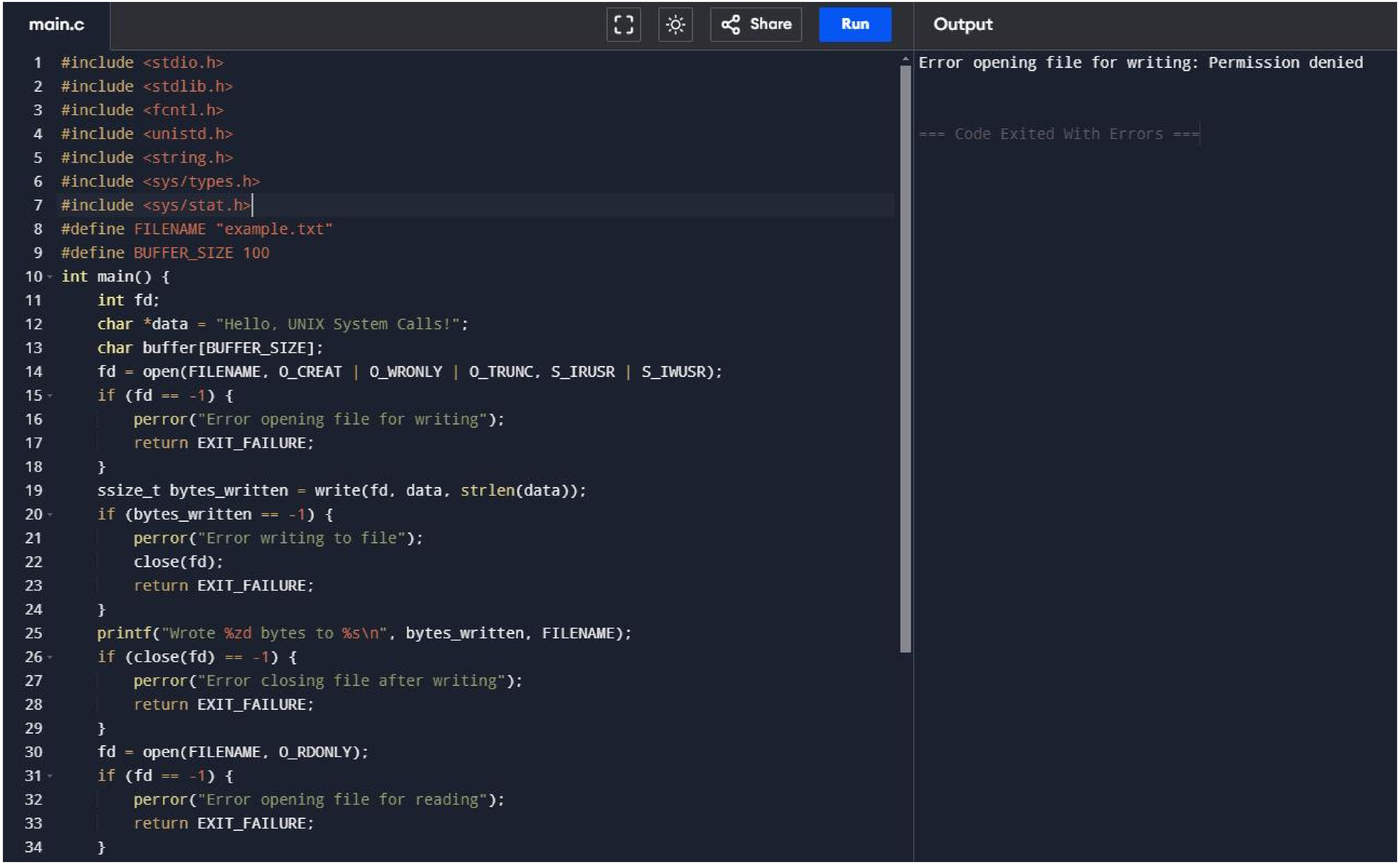


```c
#include <stdio.h>
#define MAX_BLOCKS 100
void firstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }
    printf("Process No.\tBlock No.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t", i + 1);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}
int main() {
    int blockSize[MAX_BLOCKS] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);
    firstFit(blockSize, m, processSize, n);
    return 0;
}
```

```
Process No. Block No.
1       2
2       5
3       2
4       Not Allocated


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

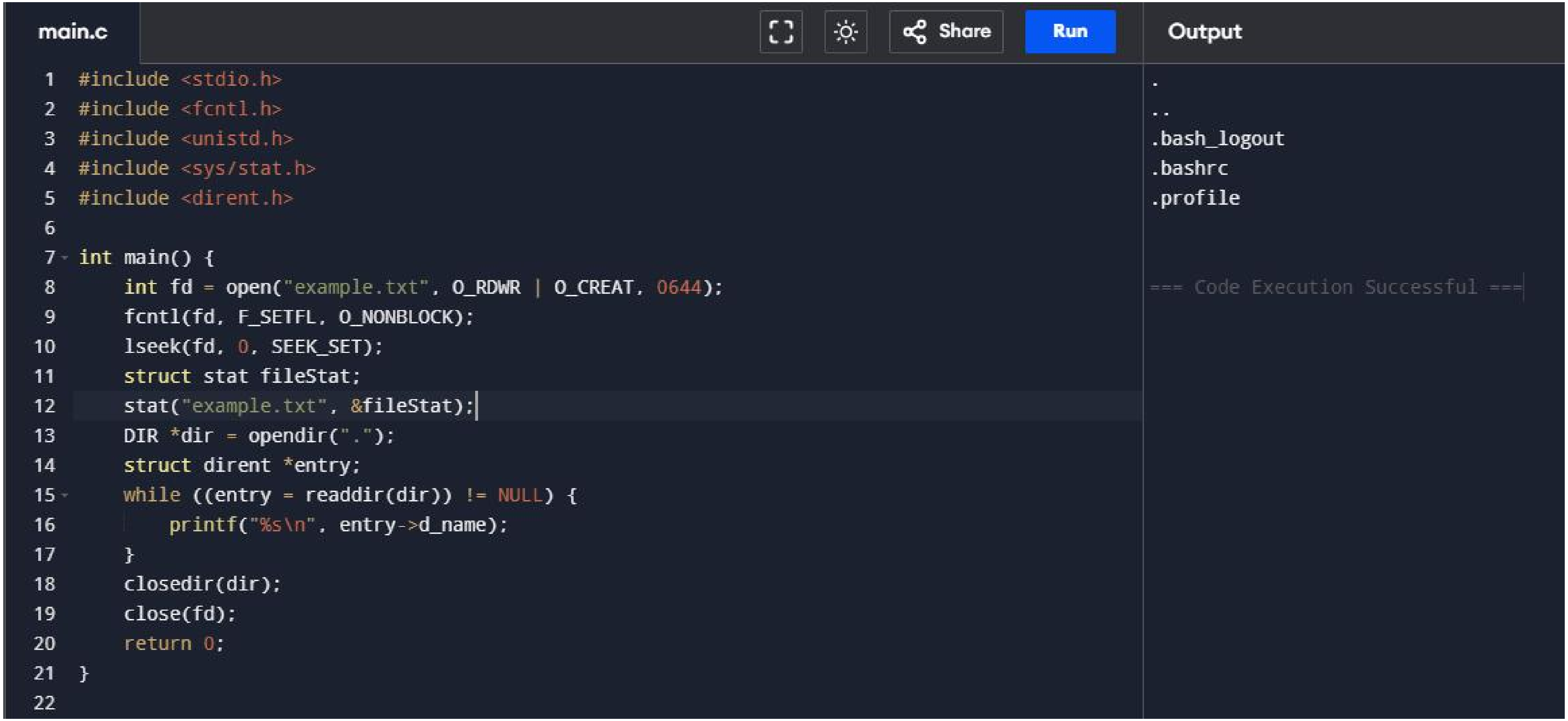


```c
    ssize_t bytes_read = read(fd, buffer, BUFFER_SIZE - 1);
    if (bytes_read == -1) {
        perror("Error reading from file");
        close(fd);
        return EXIT_FAILURE;
    }
    buffer[bytes_read] = '\0';
    printf("Read %zd bytes from %s: %s\n", bytes_read, FILENAME, buffer);
    if (close(fd) == -1) {
        perror("Error closing file after reading");
        return EXIT_FAILURE;
    }
    if (unlink(FILENAME) == -1) {
        perror("Error deleting file");
        return EXIT_FAILURE;
    }
    printf("Deleted file %s\n", FILENAME);
    return EXIT_SUCCESS;
}
```



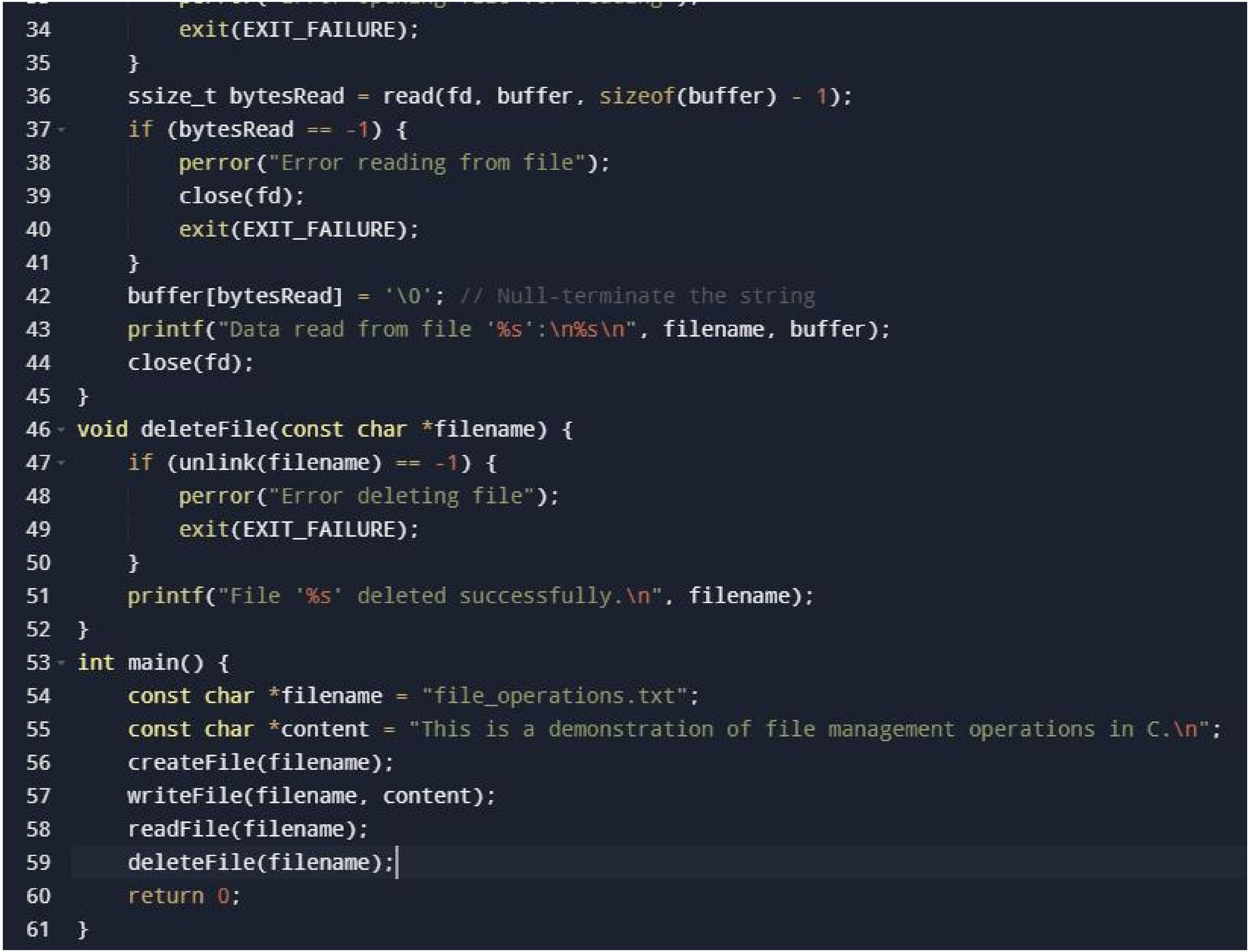
```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FILENAME "example.txt"
#define BUFFER_SIZE 100
int main() {
    int fd;
    char *data = "Hello, UNIX System Calls!";
    char buffer[BUFFER_SIZE];
    fd = open(FILENAME, O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("Error opening file for writing");
        return EXIT_FAILURE;
    }
    ssize_t bytes_written = write(fd, data, strlen(data));
    if (bytes_written == -1) {
        perror("Error writing to file");
        close(fd);
        return EXIT_FAILURE;
    }
    printf("Wrote %zd bytes to %s\n", bytes_written, FILENAME);
    if (close(fd) == -1) {
        perror("Error closing file after writing");
        return EXIT_FAILURE;
    }
    fd = open(FILENAME, O_RDONLY);
    if (fd == -1) {
        perror("Error opening file for reading");
        return EXIT_FAILURE;
    }
```

```
Error opening file for writing: Permission denied


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

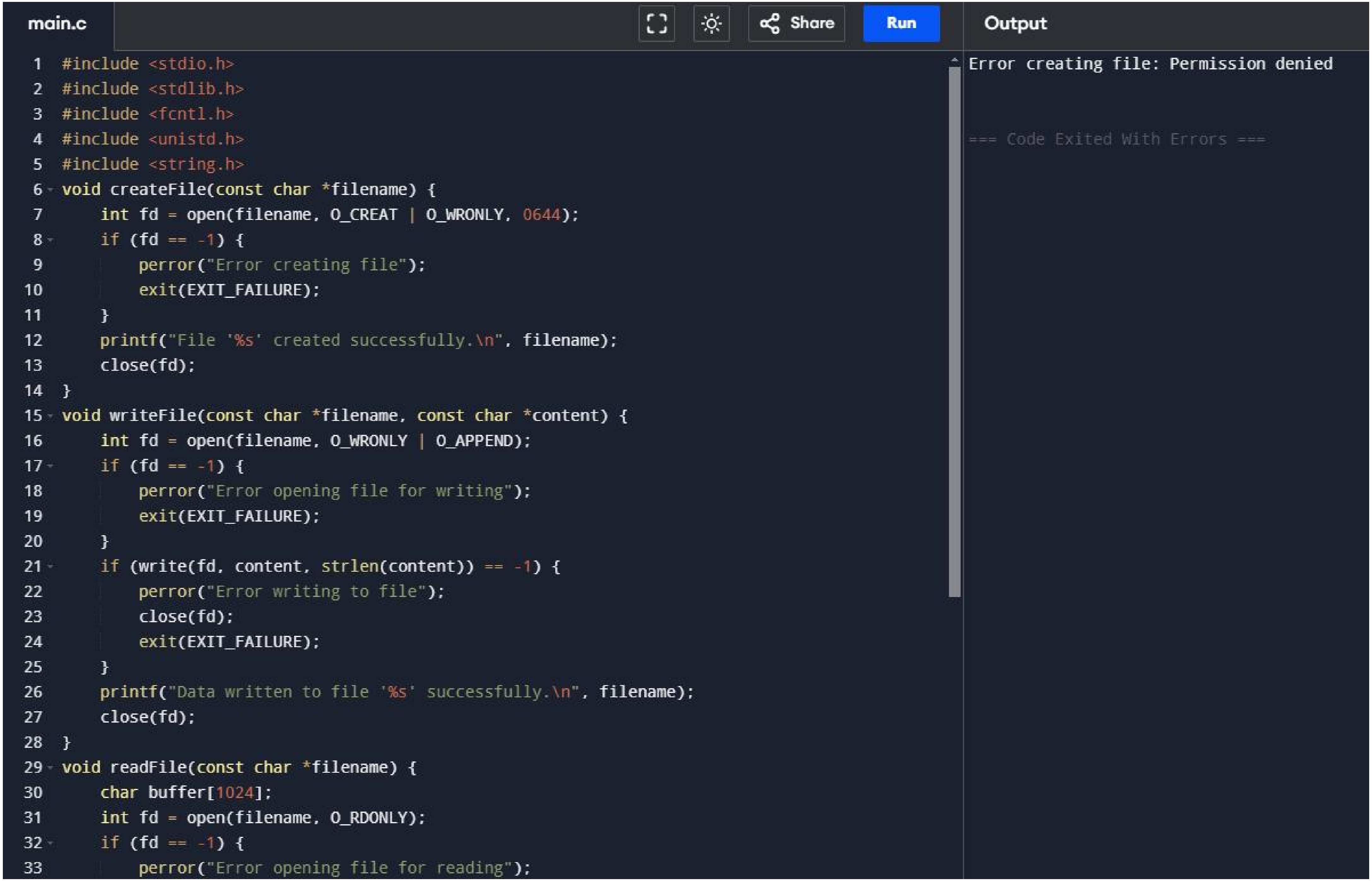


```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <dirent.h>

int main() {
    int fd = open("example.txt", O_RDWR | O_CREAT, 0644);
    fcntl(fd, F_SETFL, O_NONBLOCK);
    lseek(fd, 0, SEEK_SET);
    struct stat fileStat;
    stat("example.txt", &fileStat);
    DIR *dir = opendir(".");
    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }
    closedir(dir);
    close(fd);
    return 0;
}

```

```
.
..
.bash_logout
.bashrc
.profile


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

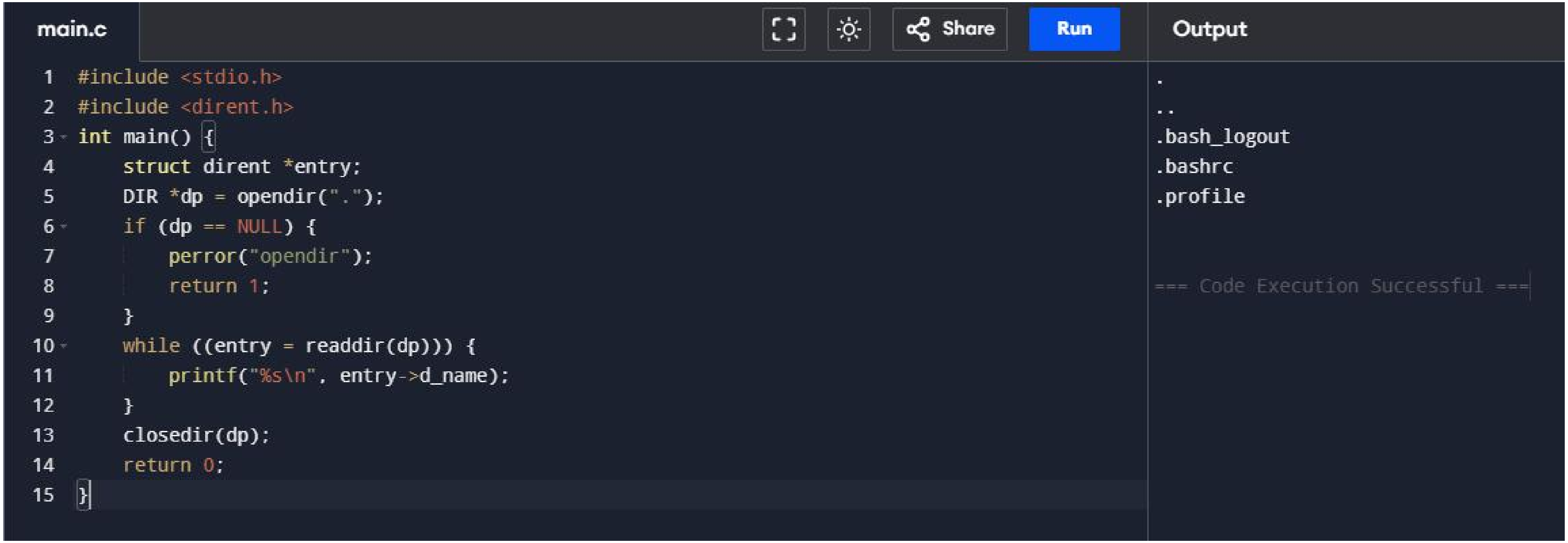


```c
        perror("Error opening file for reading");
        exit(EXIT_FAILURE);
    }
    ssize_t bytesRead = read(fd, buffer, sizeof(buffer) - 1);
    if (bytesRead == -1) {
        perror("Error reading from file");
        close(fd);
        exit(EXIT_FAILURE);
    }
    buffer[bytesRead] = '\0'; // Null-terminate the string
    printf("Data read from file '%s':\n%s\n", filename, buffer);
    close(fd);
}
void deleteFile(const char *filename) {
    if (unlink(filename) == -1) {
        perror("Error deleting file");
        exit(EXIT_FAILURE);
    }
    printf("File '%s' deleted successfully.\n", filename);
}
int main() {
    const char *filename = "file_operations.txt";
    const char *content = "This is a demonstration of file management operations in C.\n";
    createFile(filename);
    writeFile(filename, content);
    readFile(filename);
    deleteFile(filename);
    return 0;
}
```



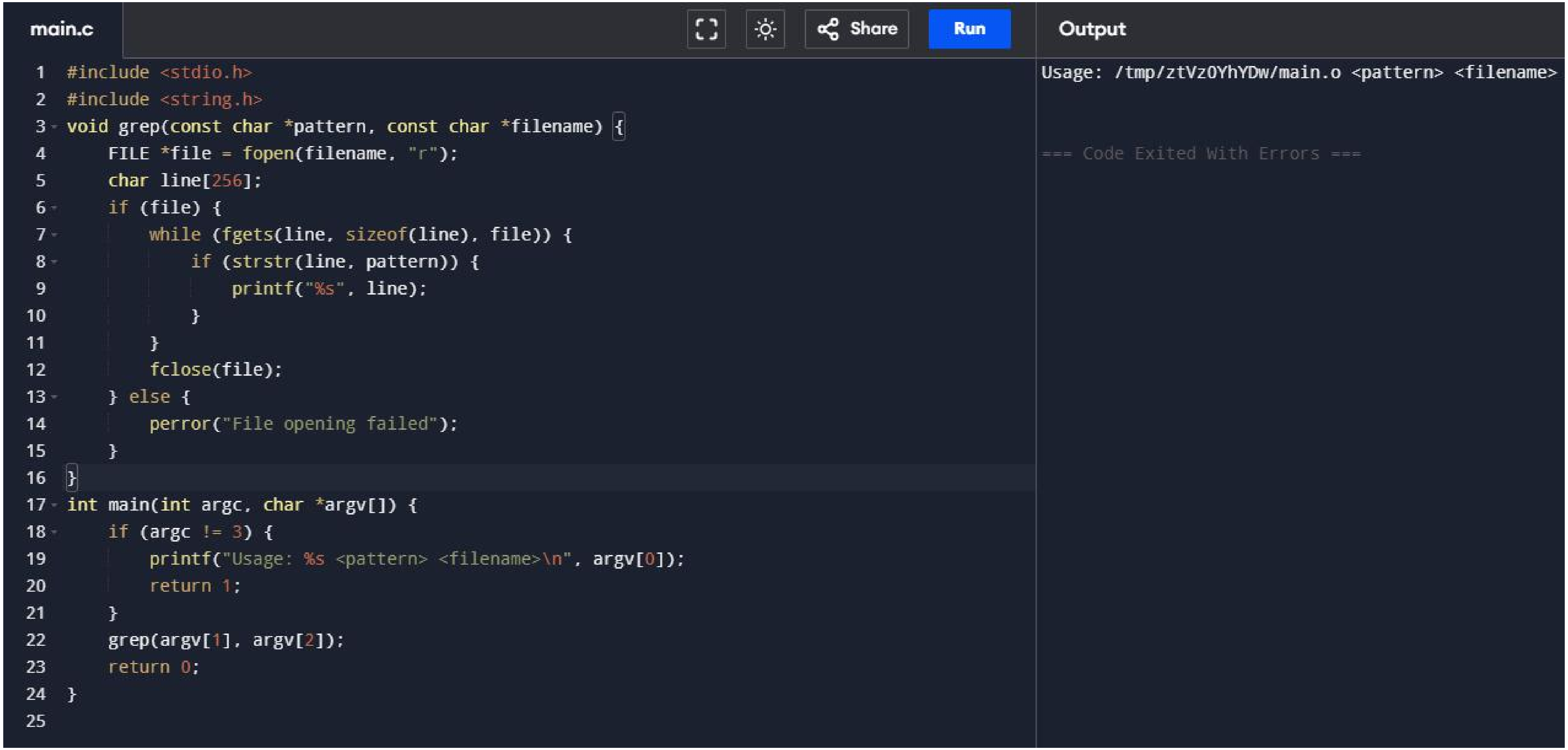
```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
void createFile(const char *filename) {
    int fd = open(filename, O_CREAT | O_WRONLY, 0644);
    if (fd == -1) {
        perror("Error creating file");
        exit(EXIT_FAILURE);
    }
    printf("File '%s' created successfully.\n", filename);
    close(fd);
}
void writeFile(const char *filename, const char *content) {
    int fd = open(filename, O_WRONLY | O_APPEND);
    if (fd == -1) {
        perror("Error opening file for writing");
        exit(EXIT_FAILURE);
    }
    if (write(fd, content, strlen(content)) == -1) {
        perror("Error writing to file");
        close(fd);
        exit(EXIT_FAILURE);
    }
    printf("Data written to file '%s' successfully.\n", filename);
    close(fd);
}
void readFile(const char *filename) {
    char buffer[1024];
    int fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror("Error opening file for reading");
```

```
Error creating file: Permission denied


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



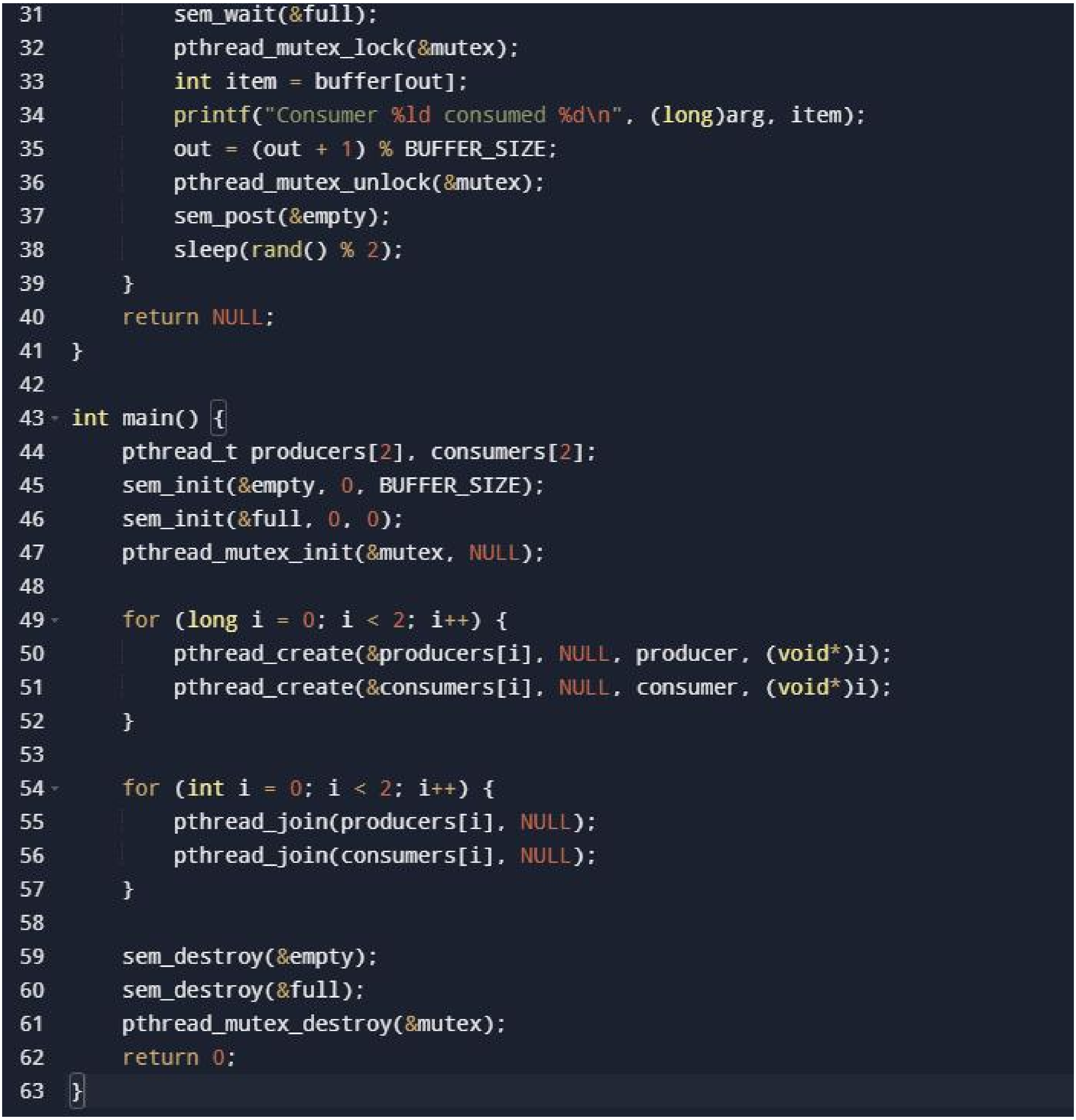
```c
#include <stdio.h>
#include <dirent.h>
int main() {
    struct dirent *entry;
    DIR *dp = opendir(".");
    if (dp == NULL) {
        perror("opendir");
        return 1;
    }
    while ((entry = readdir(dp))) {
        printf("%s\n", entry->d_name);
    }
    closedir(dp);
    return 0;
}
```

```
.
..
.bash_logout
.bashrc
.profile


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

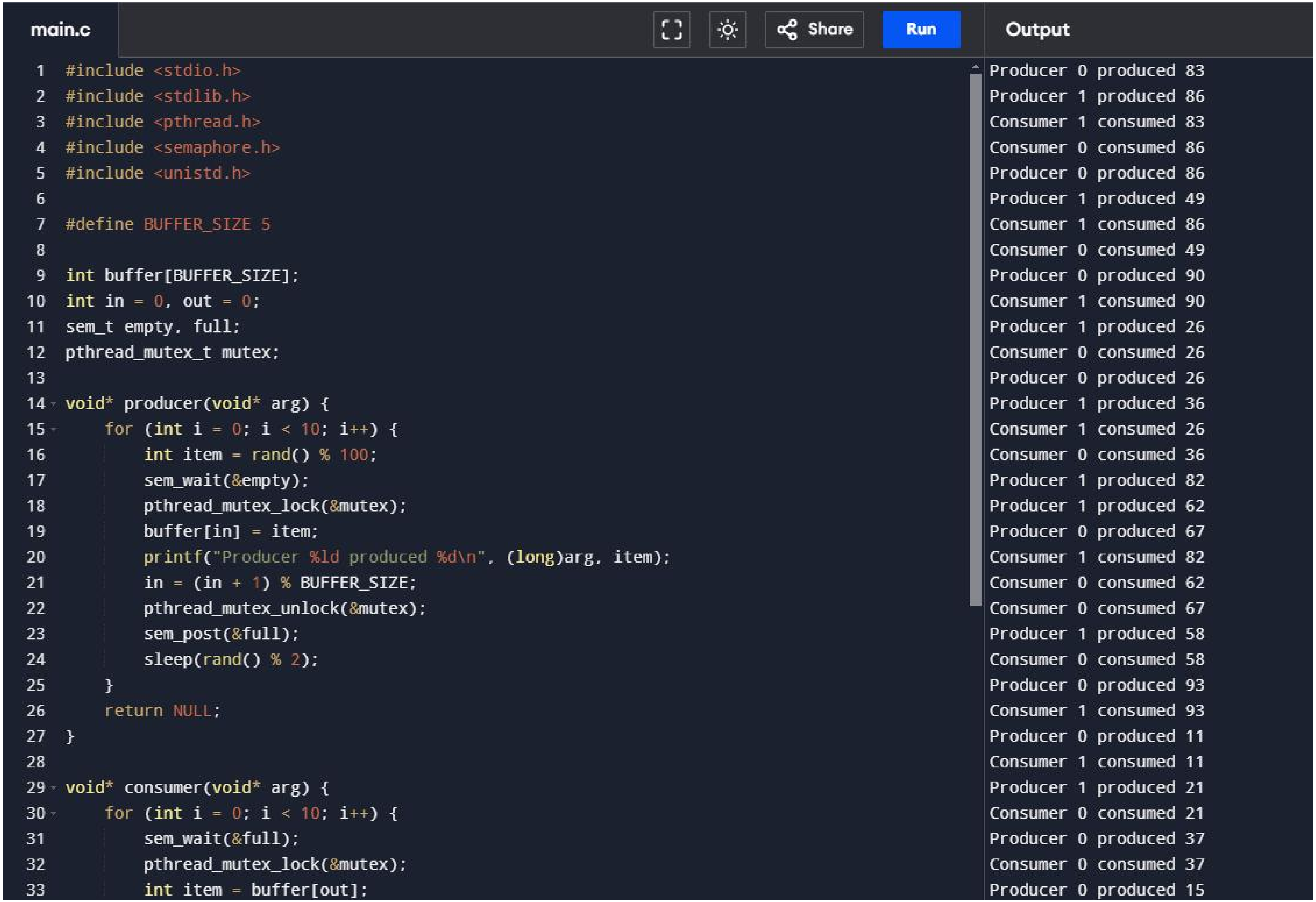


```c
#include <stdio.h>
#include <string.h>
void grep(const char *pattern, const char *filename) {
    FILE *file = fopen(filename, "r");
    char line[256];
    if (file) {
        while (fgets(line, sizeof(line), file)) {
            if (strstr(line, pattern)) {
                printf("%s", line);
            }
        }
        fclose(file);
    } else {
        perror("File opening failed");
    }
}
int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <pattern> <filename>\n", argv[0]);
        return 1;
    }
    grep(argv[1], argv[2]);
    return 0;
}

```

```
Usage: /tmp/ztVzOYhYDw/main.o <pattern> <filename>


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

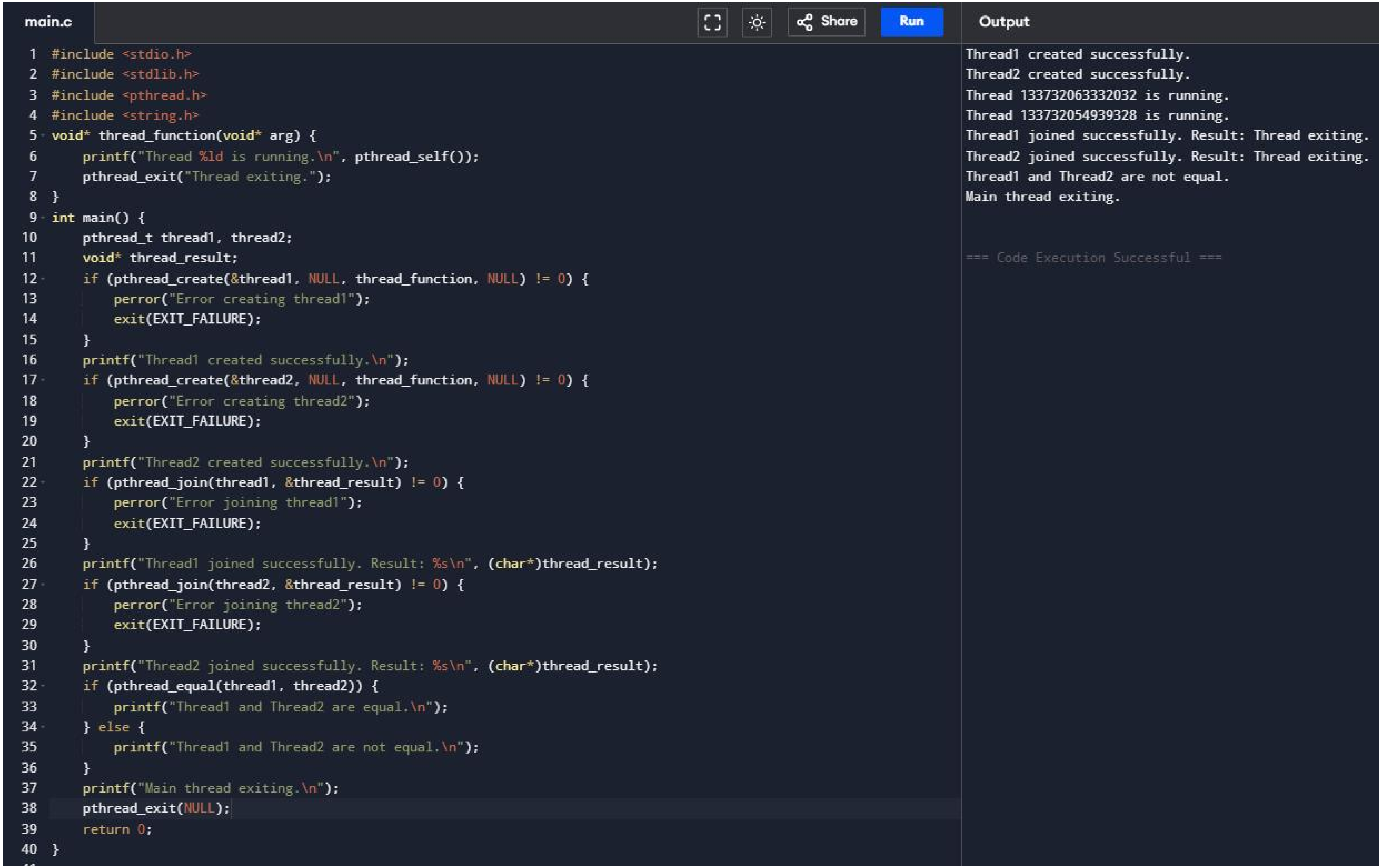


```c
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        printf("Consumer %ld consumed %d\n", (long)arg, item);
        out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
        sleep(rand() % 2);
    }
    return NULL;
}

int main() {
    pthread_t producers[2], consumers[2];
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    for (long i = 0; i < 2; i++) {
        pthread_create(&producers[i], NULL, producer, (void*)i);
        pthread_create(&consumers[i], NULL, consumer, (void*)i);
    }

    for (int i = 0; i < 2; i++) {
        pthread_join(producers[i], NULL);
        pthread_join(consumers[i], NULL);
    }

    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```



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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0;
sem_t empty, full;
pthread_mutex_t mutex;

void* producer(void* arg) {
    for (int i = 0; i < 10; i++) {
        int item = rand() % 100;
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        buffer[in] = item;
        printf("Producer %ld produced %d\n", (long)arg, item);
        in = (in + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
        sleep(rand() % 2);
    }
    return NULL;
}

void* consumer(void* arg) {
    for (int i = 0; i < 10; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
```

```
Producer 0 produced 83
Producer 1 produced 86
Consumer 1 consumed 83
Consumer 0 consumed 86
Producer 0 produced 86
Producer 1 produced 49
Consumer 1 consumed 86
Consumer 0 consumed 49
Producer 0 produced 90
Consumer 1 consumed 90
Producer 1 produced 26
Consumer 0 consumed 26
Producer 0 produced 26
Producer 1 produced 36
Consumer 1 consumed 26
Consumer 0 consumed 36
Producer 1 produced 82
Producer 1 produced 62
Producer 0 produced 67
Consumer 1 consumed 82
Consumer 0 consumed 62
Consumer 0 consumed 67
Producer 1 produced 58
Consumer 0 consumed 58
Producer 0 produced 93
Consumer 1 consumed 93
Producer 0 produced 11
Consumer 1 consumed 11
Producer 1 produced 21
Consumer 0 consumed 21
Producer 0 produced 37
Consumer 0 consumed 37
Producer 0 produced 15
```



```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
void* thread_function(void* arg) {
    printf("Thread %ld is running.\n", pthread_self());
    pthread_exit("Thread exiting.");
}
int main() {
    pthread_t thread1, thread2;
    void* thread_result;
    if (pthread_create(&thread1, NULL, thread_function, NULL) != 0) {
        perror("Error creating thread1");
        exit(EXIT_FAILURE);
    }
    printf("Thread1 created successfully.\n");
    if (pthread_create(&thread2, NULL, thread_function, NULL) != 0) {
        perror("Error creating thread2");
        exit(EXIT_FAILURE);
    }
    printf("Thread2 created successfully.\n");
    if (pthread_join(thread1, &thread_result) != 0) {
        perror("Error joining thread1");
        exit(EXIT_FAILURE);
    }
    printf("Thread1 joined successfully. Result: %s\n", (char*)thread_result);
    if (pthread_join(thread2, &thread_result) != 0) {
        perror("Error joining thread2");
        exit(EXIT_FAILURE);
    }
    printf("Thread2 joined successfully. Result: %s\n", (char*)thread_result);
    if (pthread_equal(thread1, thread2)) {
        printf("Thread1 and Thread2 are equal.\n");
    } else {
        printf("Thread1 and Thread2 are not equal.\n");
    }
    printf("Main thread exiting.\n");
    pthread_exit(NULL);
    return 0;
}
```

```
Thread1 created successfully.
Thread2 created successfully.
Thread 133732063332032 is running.
Thread 133732054939328 is running.
Thread1 joined successfully. Result: Thread exiting.
Thread2 joined successfully. Result: Thread exiting.
Thread1 and Thread2 are not equal.
Main thread exiting.


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