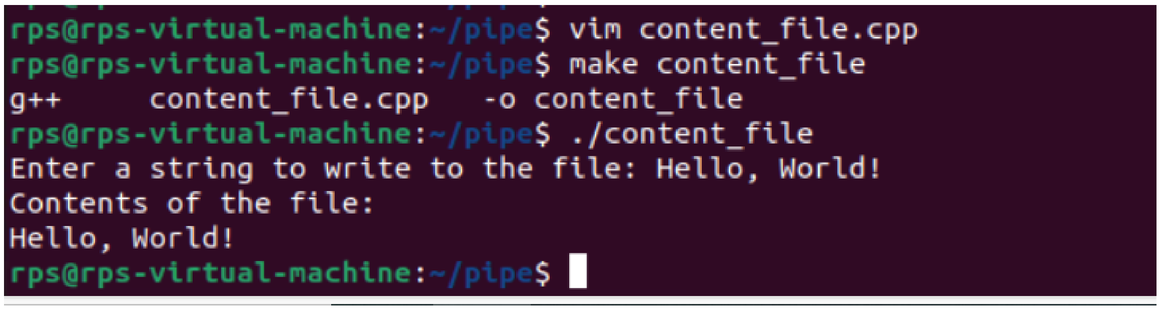
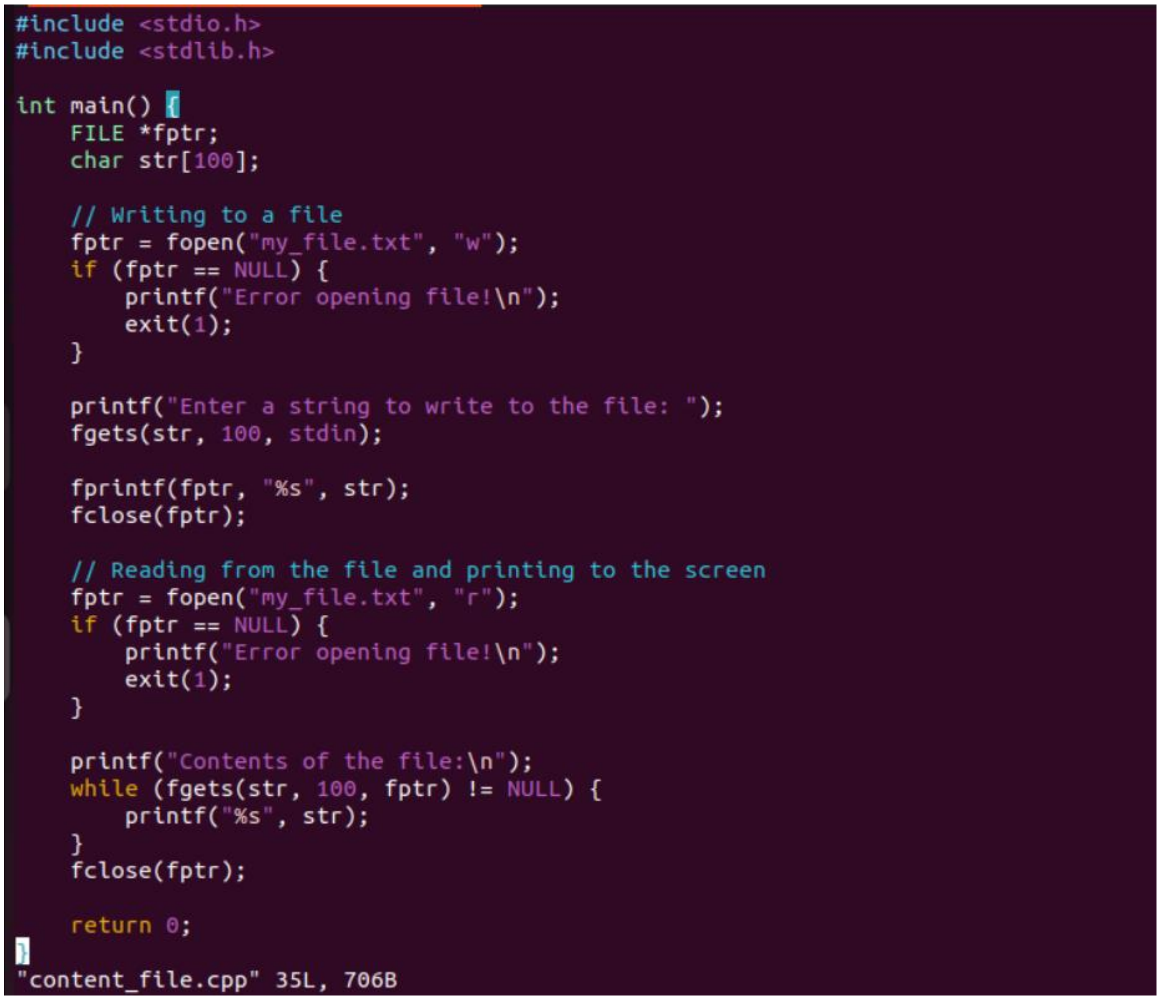
DATE :- 01-08-24

PROGRAM TO PRINT CONTENT OF THE FILE (C) :-

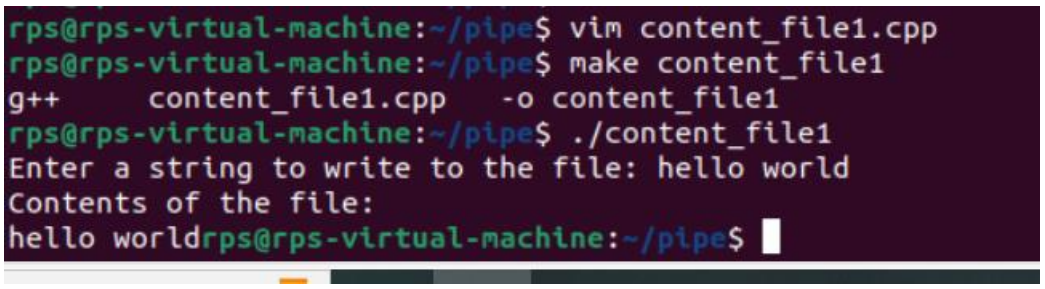


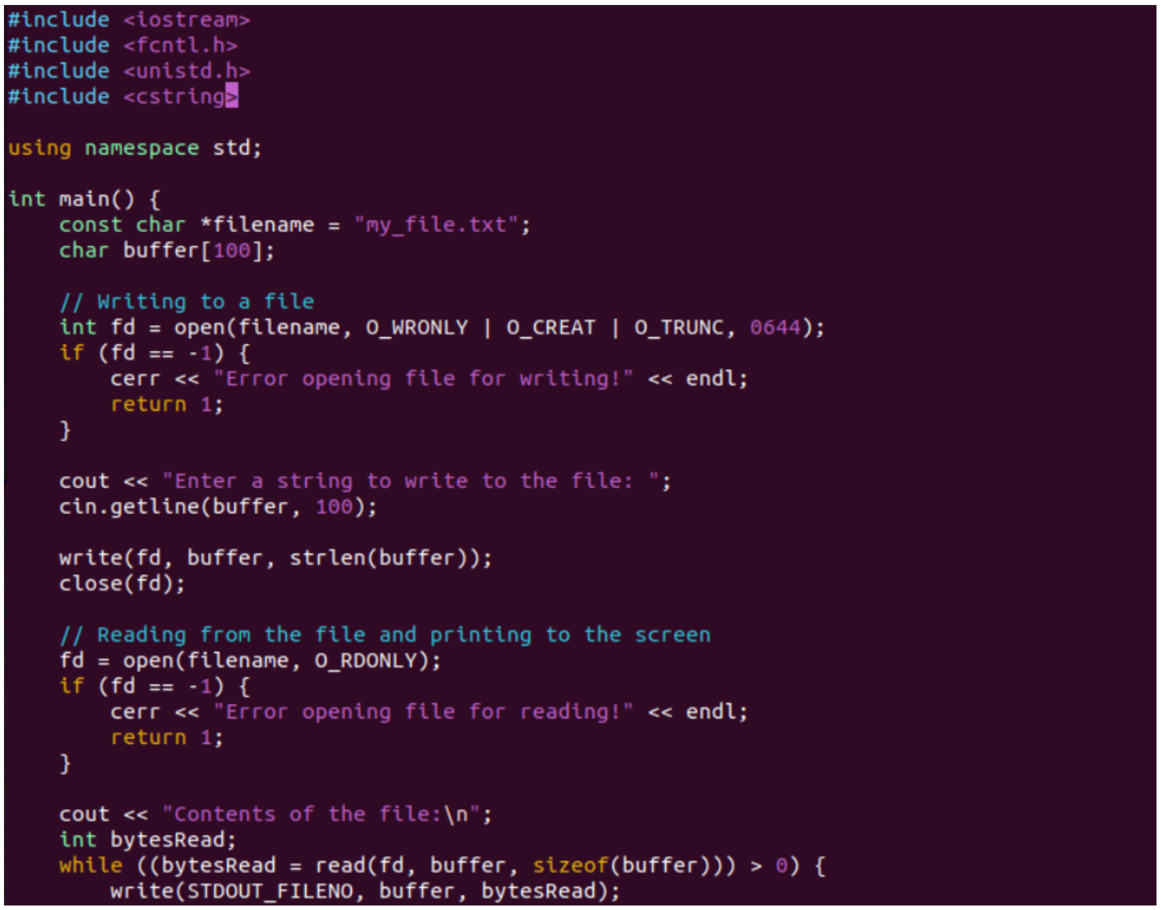
```
rps@rps-virtual-machine:~/pipe$ vim content_file.cpp
rps@rps-virtual-machine:~/pipe$ make content_file
g++     content_file.cpp   -o content_file
rps@rps-virtual-machine:~/pipe$ ./content_file
Enter a string to write to the file: Hello, World!
Contents of the file:
Hello, World!
rps@rps-virtual-machine:~/pipe$ 
```

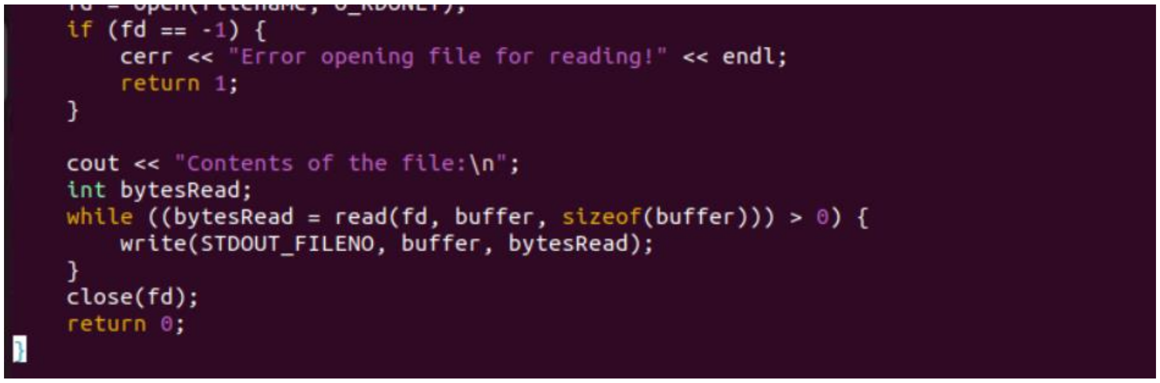


```
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *fptr;
    char str[100];

    // Writing to a file
    fptr = fopen("my_file.txt", "w");
    if (fptr == NULL) {
        printf("Error opening file!\n");
        exit(1);
    }

    printf("Enter a string to write to the file: ");
    fgets(str, 100, stdin);

    fprintf(fptr, "%s", str);
    fclose(fptr);

    // Reading from the file and printing to the screen
    fptr = fopen("my_file.txt", "r");
    if (fptr == NULL) {
        printf("Error opening file!\n");
        exit(1);
    }

    printf("Contents of the file:\n");
    while (fgets(str, 100, fptr) != NULL) {
        printf("%s", str);
    }
    fclose(fptr);

    return 0;
}
"content_file.cpp" 35L, 706B
```

PROGRAM TO PRINT CONTENT OF THE FILE (C++)   :-


```
rps@rps-virtual-machine:~/pipe$ vim content_file1.cpp
rps@rps-virtual-machine:~/pipe$ make content_file1
g++     content_file1.cpp   -o content_file1
rps@rps-virtual-machine:~/pipe$ ./content_file1
Enter a string to write to the file: hello world
Contents of the file:
hello worldrps@rps-virtual-machine:~/pipe$ 
```


```cpp
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include <cstring>

using namespace std;

int main() {
    const char *filename = "my_file.txt";
    char buffer[100];

    // Writing to a file
    int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        cerr << "Error opening file for writing!" << endl;
        return 1;
    }

    cout << "Enter a string to write to the file: ";
    cin.getline(buffer, 100);

    write(fd, buffer, strlen(buffer));
    close(fd);

    // Reading from the file and printing to the screen
    fd = open(filename, O_RDONLY);
    if (fd == -1) {
        cerr << "Error opening file for reading!" << endl;
        return 1;
    }

    cout << "Contents of the file:\n";
    int bytesRead;
    while ((bytesRead = read(fd, buffer, sizeof(buffer))) > 0) {
        write(STDOUT_FILENO, buffer, bytesRead);
```



```
    if (fd == -1) {
        cerr << "Error opening file for reading!" << endl;
        return 1;
    }

    cout << "Contents of the file:\n";
    int bytesRead;
    while ((bytesRead = read(fd, buffer, sizeof(buffer))) > 0) {
        write(STDOUT_FILENO, buffer, bytesRead);
    }
    close(fd);
    return 0;
}
```

Develop a C++ application that utilizes system calls to perform basic file I/O operations.

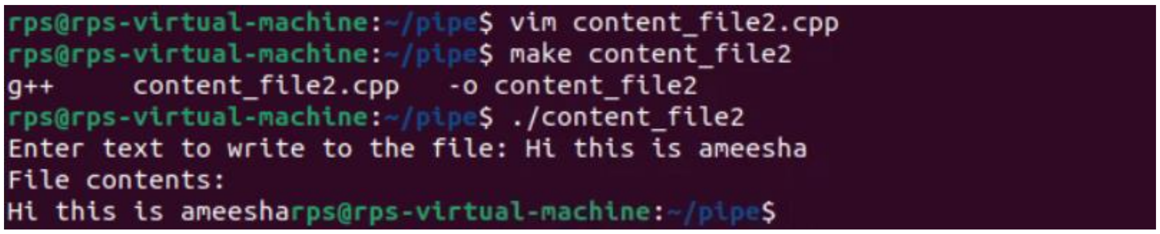
Specific Requirements:

Create a new file if it doesn't exist.

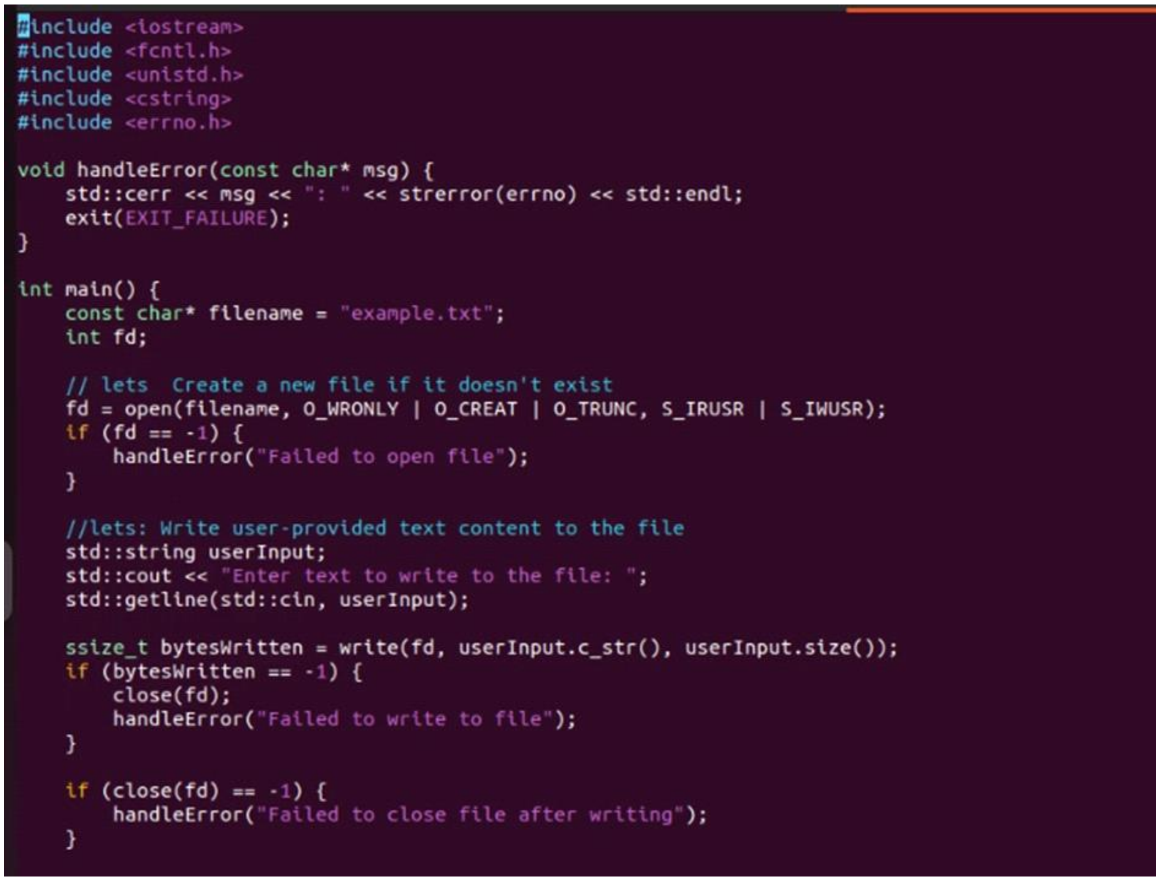
Write user-provided text content to the file.

Read the contents of the file and display them on the console.

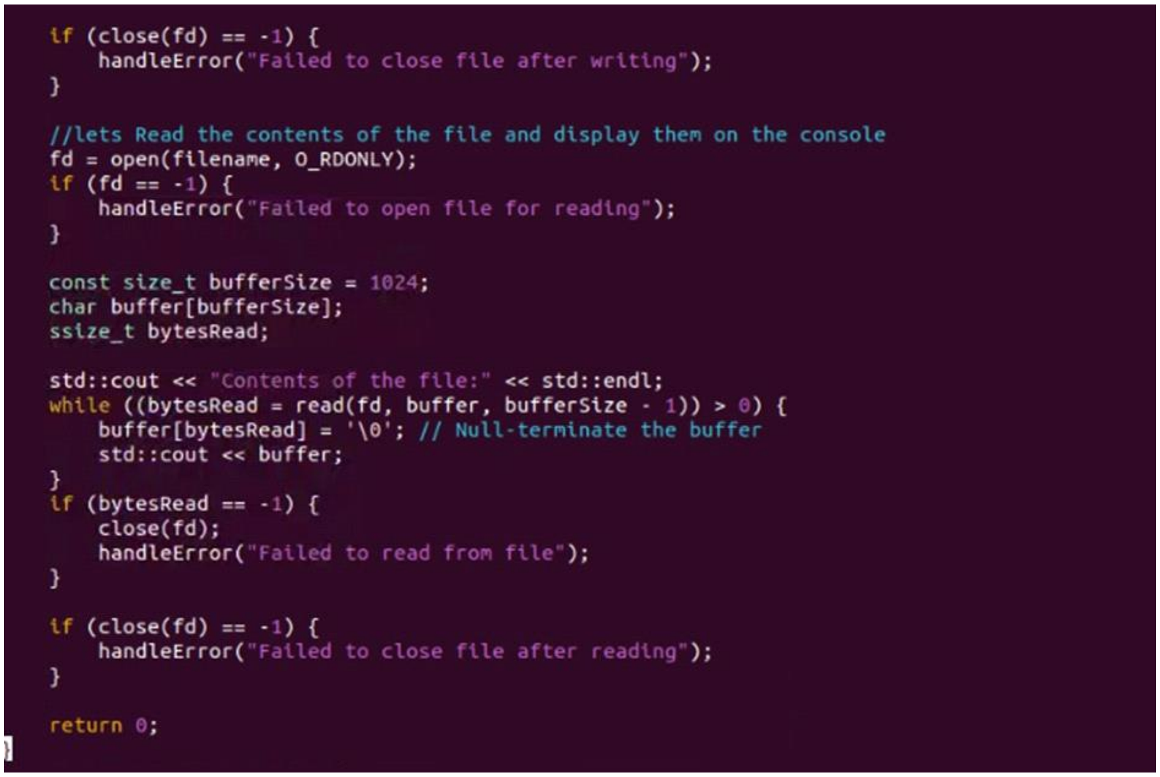
Implement robust error handling for file operations.



```
rps@rps-virtual-machine:~/pipe$ vim content_file2.cpp
rps@rps-virtual-machine:~/pipe$ make content_file2
g++     content_file2.cpp   -o content_file2
rps@rps-virtual-machine:~/pipe$ ./content_file2
Enter text to write to the file: Hi this is ameesha
File contents:
Hi this is ameesharps@rps-virtual-machine:~/pipe$
```



```
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include <cstring>
#include <errno.h>

void handleError(const char* msg) {
    std::cerr << msg << ": " << strerror(errno) << std::endl;
    exit(EXIT_FAILURE);
}

int main() {
    const char* filename = "example.txt";
    int fd;

    // lets  Create a new file if it doesn't exist
    fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        handleError("Failed to open file");
    }

    //lets: Write user-provided text content to the file
    std::string userInput;
    std::cout << "Enter text to write to the file: ";
    std::getline(std::cin, userInput);

    ssize_t bytesWritten = write(fd, userInput.c_str(), userInput.size());
    if (bytesWritten == -1) {
        close(fd);
        handleError("Failed to write to file");
    }

    if (close(fd) == -1) {
        handleError("Failed to close file after writing");
    }
```



```
    if (close(fd) == -1) {
        handleError("Failed to close file after writing");
    }

    //lets Read the contents of the file and display them on the console
    fd = open(filename, O_RDONLY);
    if (fd == -1) {
        handleError("Failed to open file for reading");
    }

    const size_t bufferSize = 1024;
    char buffer[bufferSize];
    ssize_t bytesRead;

    std::cout << "Contents of the file:" << std::endl;
    while ((bytesRead = read(fd, buffer, bufferSize - 1)) > 0) {
        buffer[bytesRead] = '\0'; // Null-terminate the buffer
        std::cout << buffer;
    }
    if (bytesRead == -1) {
        close(fd);
        handleError("Failed to read from file");
    }

    if (close(fd) == -1) {
        handleError("Failed to close file after reading");
    }

    return 0;
}
```

Short note on : man open, man close, man write, man read.

- **man open :-**

The open system call is used to open a file for reading or writing. It can also be used to create a new file if it doesn't exist.

Usage:

#include <fcntl.h>

int open(const char *pathname, int flags, mode_t mode);

Parameters:

- pathname: The name of the file to open.
- flags: Determines the file access mode (e.g., O_RDONLY, O_WRONLY, O_RDWR) and file creation options (e.g., O_CREAT, O_TRUNC).
- mode: Specifies the permissions to use in case a new file is created (e.g., S_IRUSR | S_IWUSR).

Returns:

- On success, returns a file descriptor (a non-negative integer).
- On failure, returns -1 and sets errno to indicate the error.

Example:

```
int fd = open("file.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
if (fd == -1) {
    perror("open");
}
```

- **man close :-**

The close system call is used to close an open file descriptor.

Usage:

```
#include <unistd.h>
int close(int fd);
```

Parameters:

- fd: The file descriptor to close.

Returns:

- On success, returns 0.
- On failure, returns -1 and sets errno to indicate the error.

Example:

```
if (close(fd) == -1) {
    perror("close");
}
```

- **man write :-**

The write system call is used to write data to an open file descriptor.

Usage:

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

Parameters:

- fd: The file descriptor to write to.
- buf: A pointer to the buffer containing the data to write.
- count: The number of bytes to write from the buffer.

Returns:

- On success, returns the number of bytes written.
- On failure, returns -1 and sets errno to indicate the error.

Example:

```
const char *text = "Hello, World!";
ssize_t bytes_written = write(fd, text, strlen(text));
if (bytes_written == -1) {
    perror("write");
}
```

- **man read :-**

The read system call is used to read data from an open file descriptor.

Usage:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Parameters:

- fd: The file descriptor to read from.
- buf: A pointer to the buffer where the read data will be stored.
- count: The number of bytes to read into the buffer.

Returns:

- On success, returns the number of bytes read (which may be less than count).
- On failure, returns -1 and sets errno to indicate the error.
- A return value of 0 indicates end-of-file (EOF).

Example:

```
char buffer[1024];
ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);
if (bytes_read == -1) {
    perror("read");
} else {
    buffer[bytes_read] = '\0'; // Null-terminate the buffer
    printf("Read: %s\n", buffer);
}
```

These system calls form the foundation for file I/O operations in POSIX-compliant operating systems. They provide low-level control over file descriptors, allowing for efficient and flexible file manipulation.

Task :

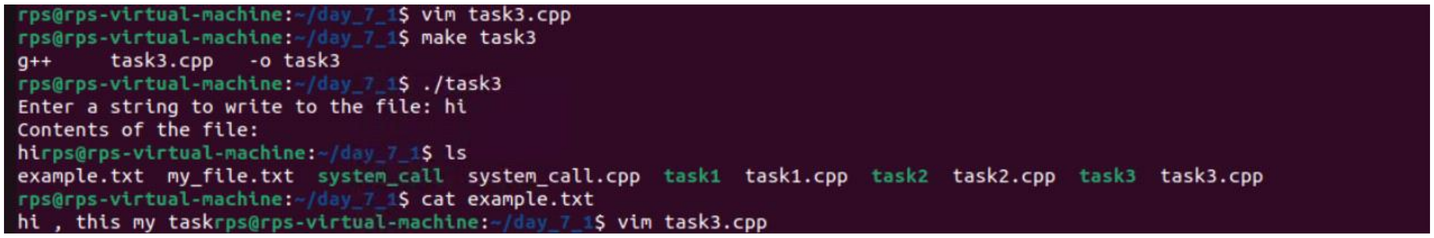
Gemini was just updated. See update

Conversation with Gemini

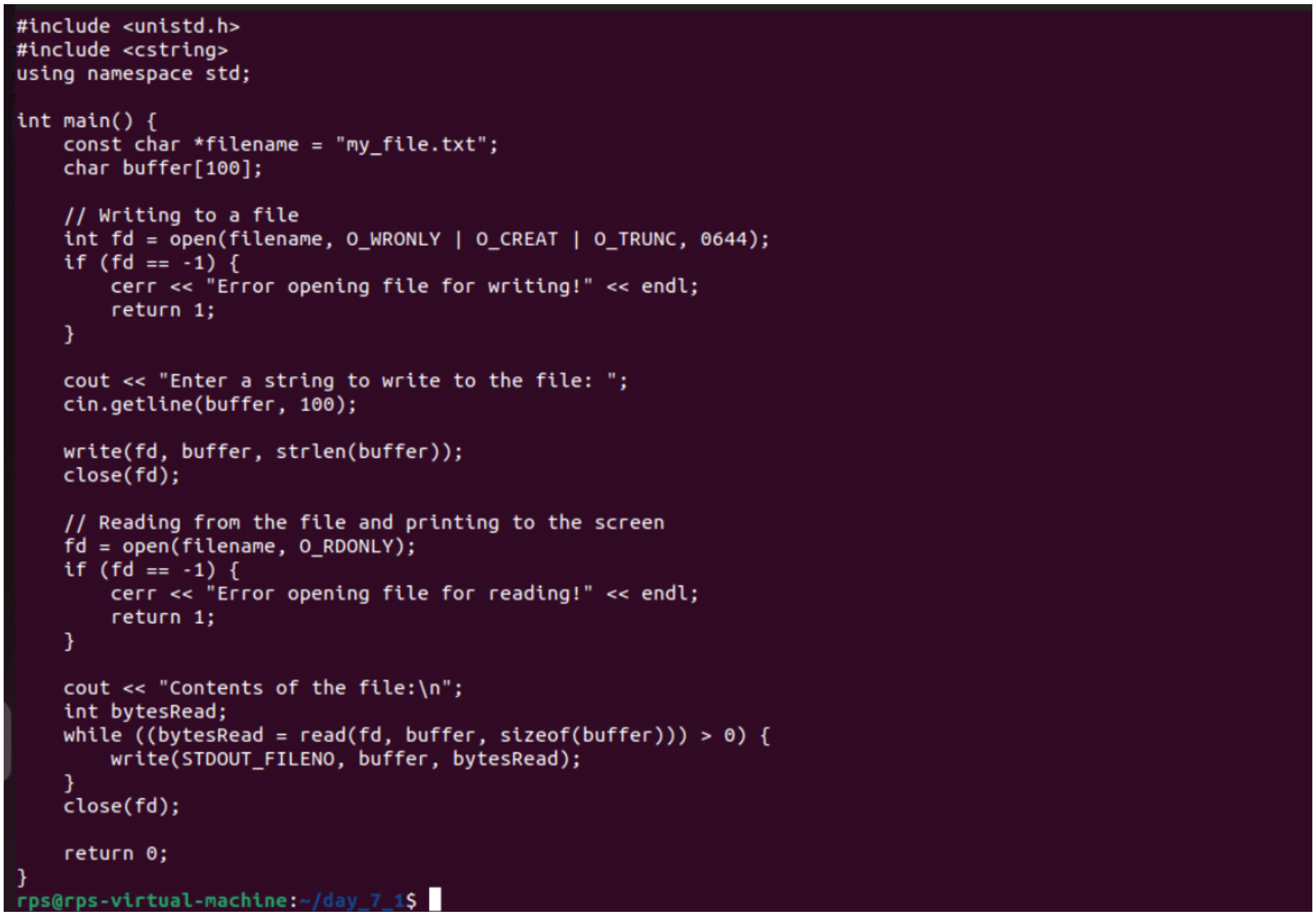
Write a c code use open close read and write to write onto file and print on screen using same

Show drafts

Code:



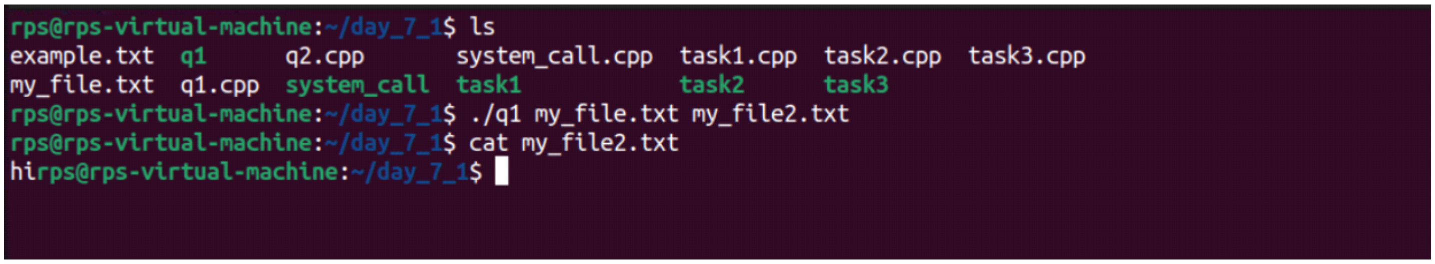
```
rps@rps-virtual-machine:~/day_7_1$ vim task3.cpp
rps@rps-virtual-machine:~/day_7_1$ make task3
g++     task3.cpp   -o task3
rps@rps-virtual-machine:~/day_7_1$ ./task3
Enter a string to write to the file: hi
Contents of the file:
hirps@rps-virtual-machine:~/day_7_1$ ls
example.txt  my_file.txt  system_call  system_call.cpp  task1  task1.cpp  task2  task2.cpp  task3  task3.cpp
rps@rps-virtual-machine:~/day_7_1$ cat example.txt
hi , this my taskrps@rps-virtual-machine:~/day_7_1$ vim task3.cpp
```



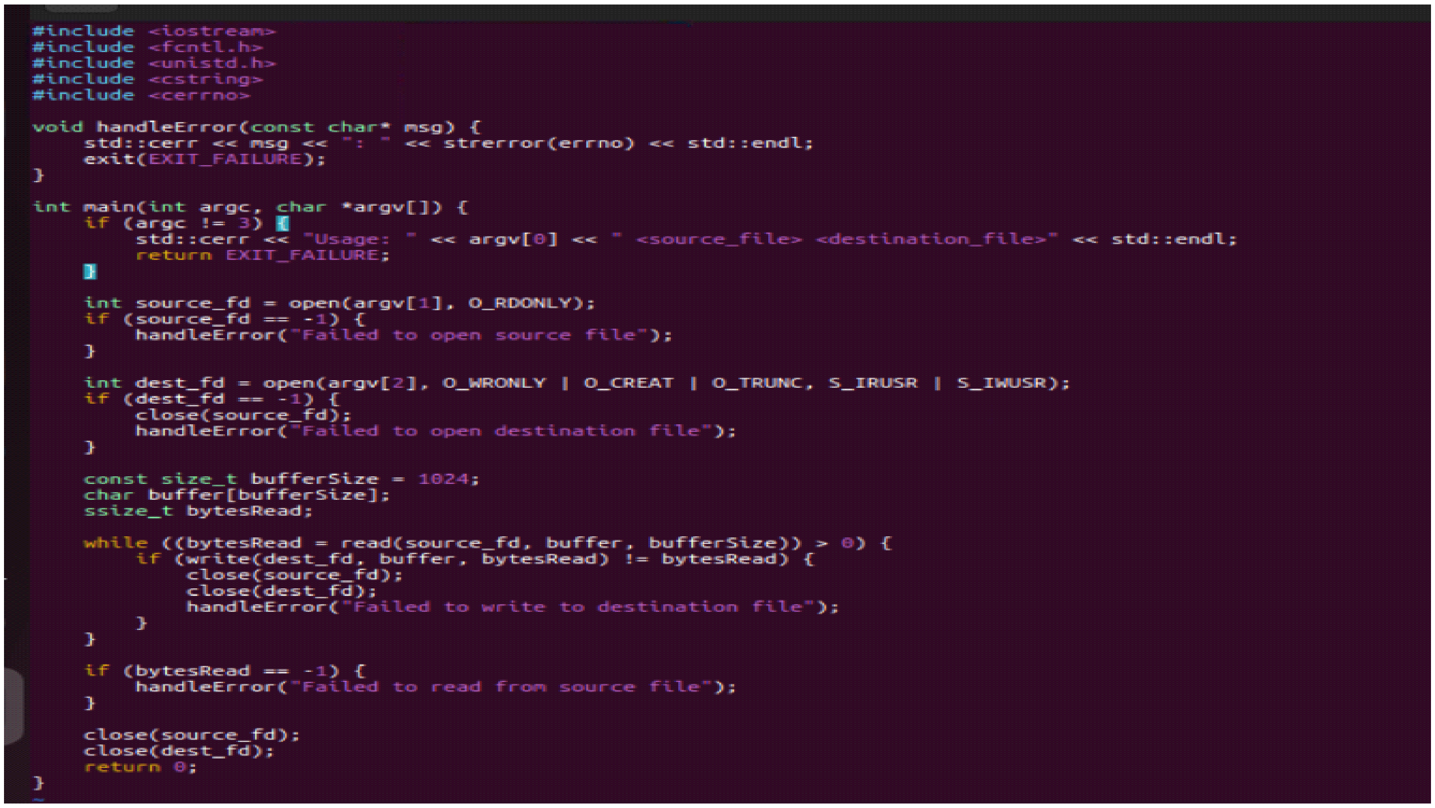
```
#include <unistd.h>
#include <cstring>
using namespace std;

int main() {
    const char *filename = "my_file.txt";
    char buffer[100];

    // Writing to a file
    int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        cerr << "Error opening file for writing!" << endl;
        return 1;
    }

    cout << "Enter a string to write to the file: ";
    cin.getline(buffer, 100);

    write(fd, buffer, strlen(buffer));
    close(fd);

    // Reading from the file and printing to the screen
    fd = open(filename, O_RDONLY);
    if (fd == -1) {
        cerr << "Error opening file for reading!" << endl;
        return 1;
    }

    cout << "Contents of the file:\n";
    int bytesRead;
    while ((bytesRead = read(fd, buffer, sizeof(buffer))) > 0) {
        write(STDOUT_FILENO, buffer, bytesRead);
    }
    close(fd);

    return 0;
}
rps@rps-virtual-machine:~/day_7_1$
```

1. File I/O and Manipulation:

Copy a File: Write a C++ program that takes two file paths as command-line arguments. Use open, read, write, and close system calls to copy the contents of the source file to the destination file. Handle potential errors (e.g., file not found, permission denied).

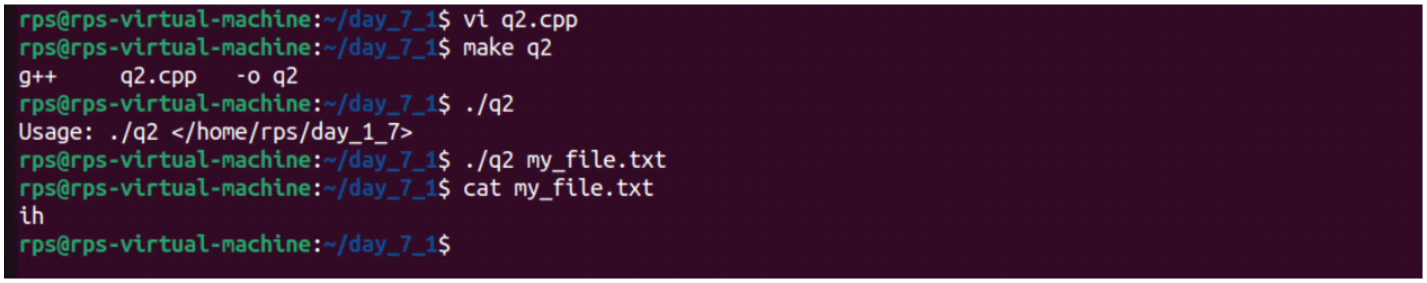


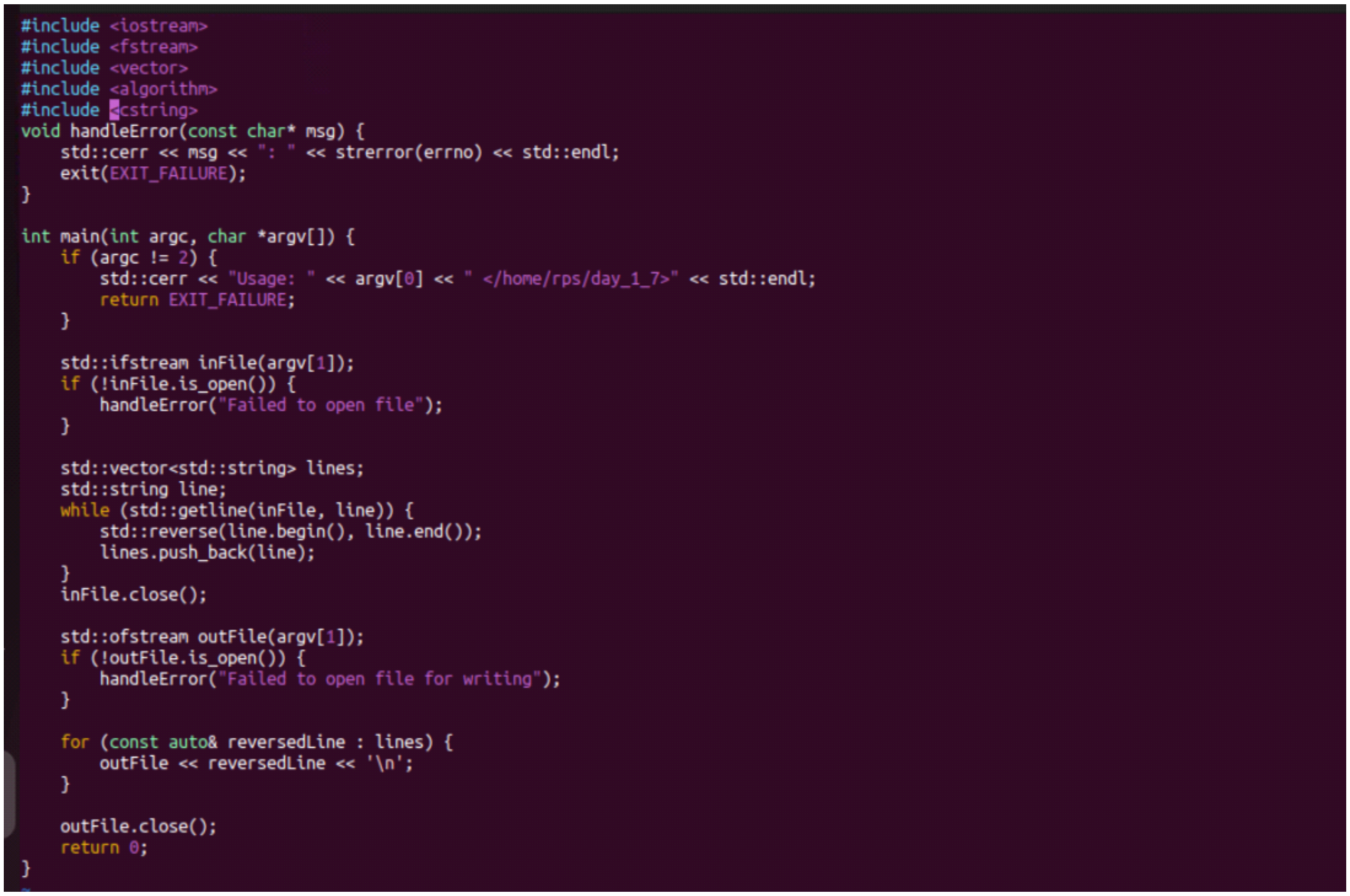
```
rps@rps-virtual-machine:~/day_7_1$ ls
example.txt  q1      q2.cpp       system_call.cpp  task1.cpp  task2.cpp  task3.cpp
my_file.txt  q1.cpp  system_call  task1            task2      task3
rps@rps-virtual-machine:~/day_7_1$ ./q1 my_file.txt my_file2.txt
rps@rps-virtual-machine:~/day_7_1$ cat my_file2.txt
hirps@rps-virtual-machine:~/day_7_1$ 
```



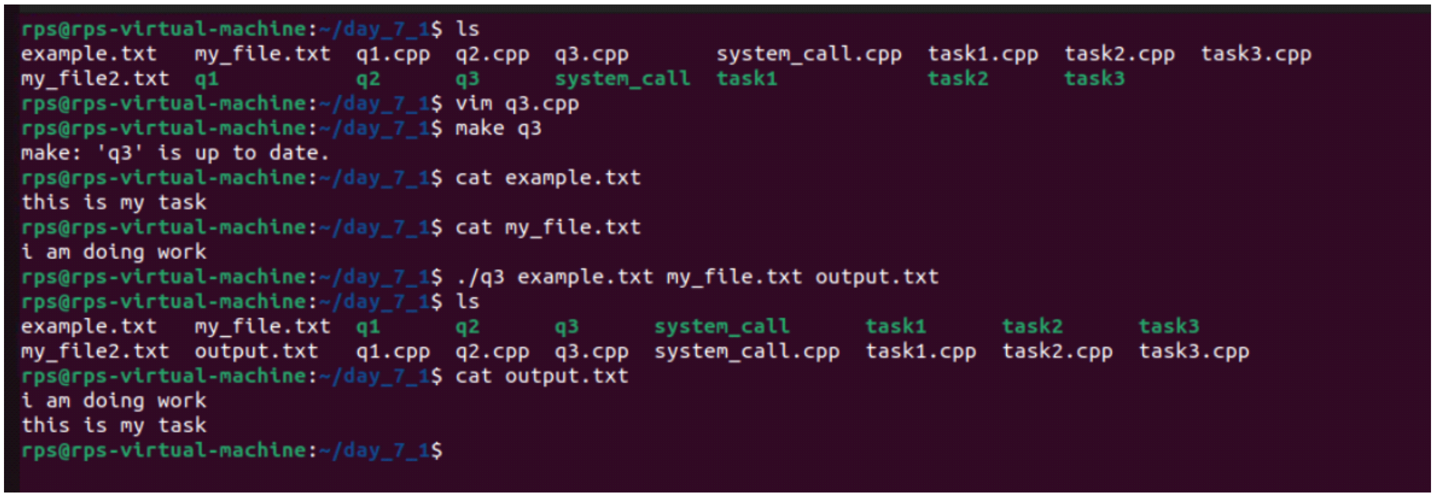
```cpp
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include <cstring>
#include <cerrno>

void handleError(const char* msg) {
    std::cerr << msg << ": " << strerror(errno) << std::endl;
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <source_file> <destination_file>" << std::endl;
        return EXIT_FAILURE;
    }

    int source_fd = open(argv[1], O_RDONLY);
    if (source_fd == -1) {
        handleError("Failed to open source file");
    }

    int dest_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if (dest_fd == -1) {
        close(source_fd);
        handleError("Failed to open destination file");
    }

    const size_t bufferSize = 1024;
    char buffer[bufferSize];
    ssize_t bytesRead;

    while ((bytesRead = read(source_fd, buffer, bufferSize)) > 0) {
        if (write(dest_fd, buffer, bytesRead) != bytesRead) {
            close(source_fd);
            close(dest_fd);
            handleError("Failed to write to destination file");
        }
    }

    if (bytesRead == -1) {
        handleError("Failed to read from source file");
    }

    close(source_fd);
    close(dest_fd);
    return 0;
}
```

2. Reverse a File: Write a C++ program that reads the contents of a file line by line, reverses each line in-place, and then writes the reversed lines back to the same file. Use system calls like open, read, write, lseek, and close to achieve this.

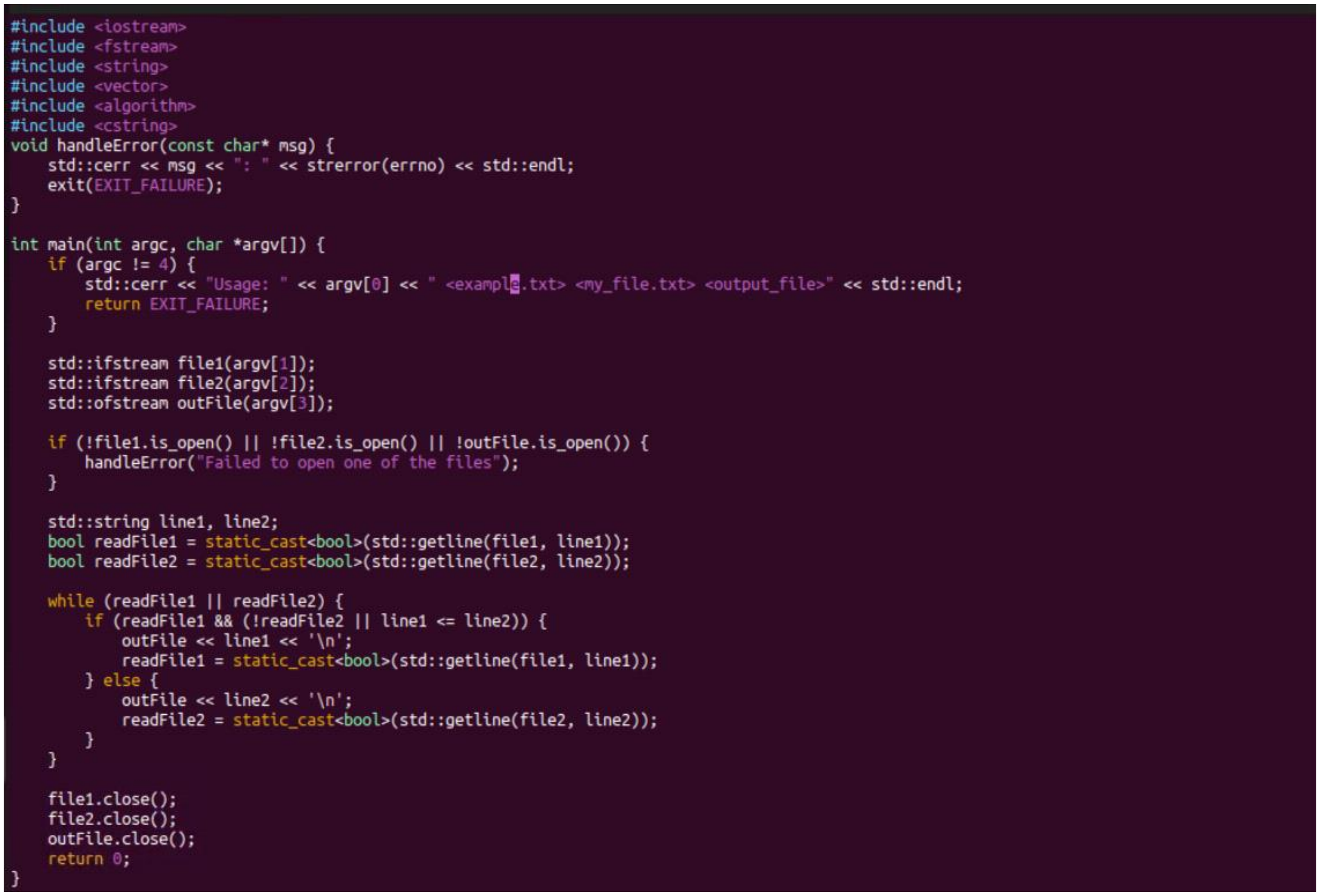

```
rps@rps-virtual-machine:~/day_7_1$ vi q2.cpp
rps@rps-virtual-machine:~/day_7_1$ make q2
g++     q2.cpp   -o q2
rps@rps-virtual-machine:~/day_7_1$ ./q2
Usage: ./q2 </home/rps/day_1_7>
rps@rps-virtual-machine:~/day_7_1$ ./q2 my_file.txt
rps@rps-virtual-machine:~/day_7_1$ cat my_file.txt
ih
rps@rps-virtual-machine:~/day_7_1$
```


```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <cstring>
void handleError(const char* msg) {
    std::cerr << msg << ": " << strerror(errno) << std::endl;
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        std::cerr << "Usage: " << argv[0] << " </home/rps/day_1_7>" << std::endl;
        return EXIT_FAILURE;
    }

    std::ifstream inFile(argv[1]);
    if (!inFile.is_open()) {
        handleError("Failed to open file");
    }

    std::vector<std::string> lines;
    std::string line;
    while (std::getline(inFile, line)) {
        std::reverse(line.begin(), line.end());
        lines.push_back(line);
    }
    inFile.close();

    std::ofstream outFile(argv[1]);
    if (!outFile.is_open()) {
        handleError("Failed to open file for writing");
    }

    for (const auto& reversedLine : lines) {
        outFile << reversedLine << '\n';
    }

    outFile.close();
    return 0;
}
```

3. Merge Two Sorted Files: Write a C++ program that takes two sorted text files as input and creates a new file containing the merged and sorted contents. Use appropriate system calls for file handling and consider memory efficiency when handling large files.



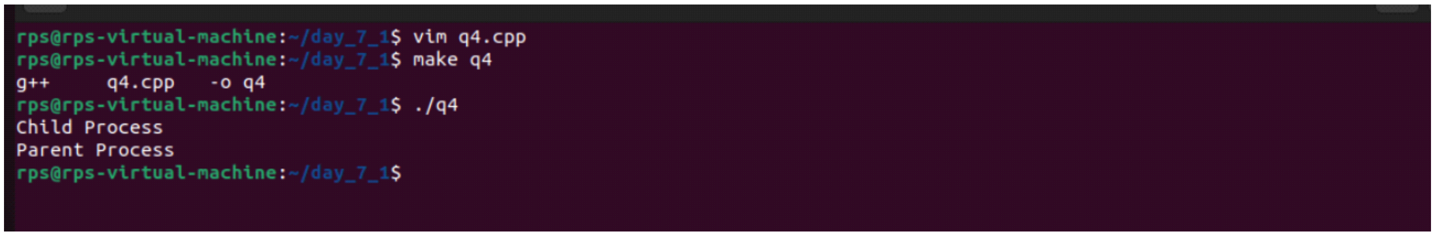
```
rps@rps-virtual-machine:~/day_7_1$ ls
example.txt   my_file.txt  q1.cpp  q2.cpp  q3.cpp       system_call.cpp  task1.cpp  task2.cpp  task3.cpp
my_file2.txt  q1           q2      q3      system_call  task1            task2      task3
rps@rps-virtual-machine:~/day_7_1$ vim q3.cpp
rps@rps-virtual-machine:~/day_7_1$ make q3
make: 'q3' is up to date.
rps@rps-virtual-machine:~/day_7_1$ cat example.txt
this is my task
rps@rps-virtual-machine:~/day_7_1$ cat my_file.txt
i am doing work
rps@rps-virtual-machine:~/day_7_1$ ./q3 example.txt my_file.txt output.txt
rps@rps-virtual-machine:~/day_7_1$ ls
example.txt   my_file.txt  q1      q2      q3      system_call      task1      task2      task3
my_file2.txt  output.txt   q1.cpp  q2.cpp  q3.cpp  system_call.cpp  task1.cpp  task2.cpp  task3.cpp
rps@rps-virtual-machine:~/day_7_1$ cat output.txt
i am doing work
this is my task
rps@rps-virtual-machine:~/day_7_1$
```



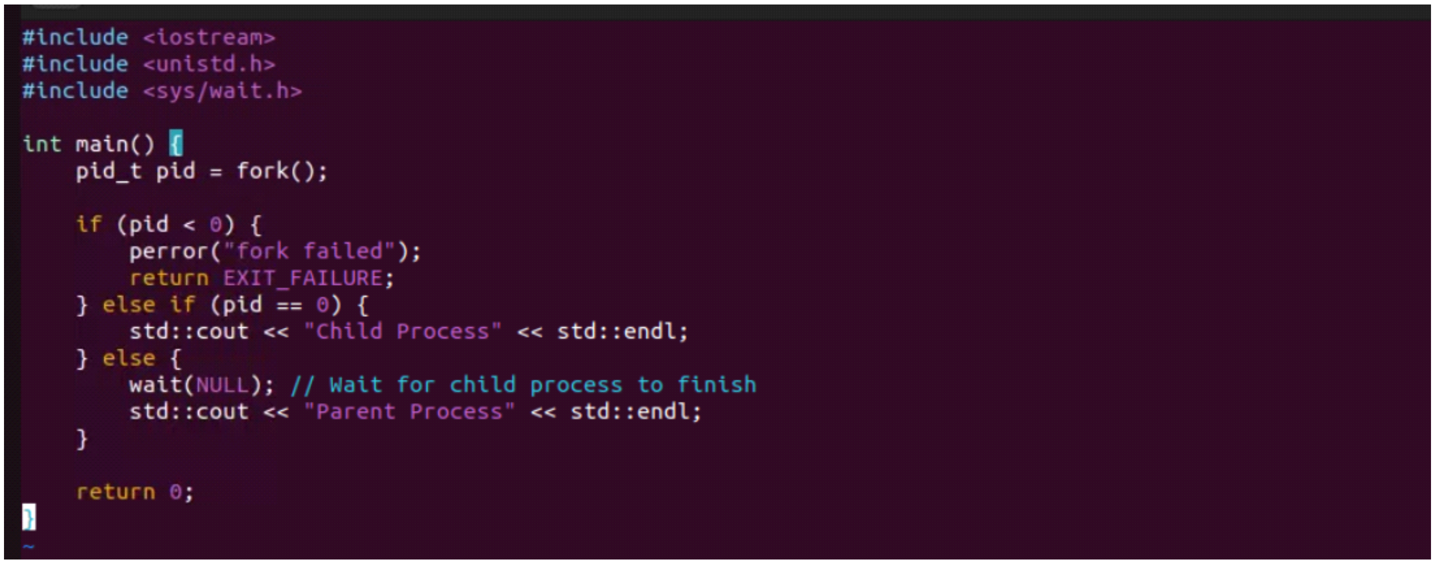
```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include <cstring>
void handleError(const char* msg) {
    std::cerr << msg << ": " << strerror(errno) << std::endl;
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
    if (argc != 4) {
        std::cerr << "Usage: " << argv[0] << " <example.txt> <my_file.txt> <output_file>" << std::endl;
        return EXIT_FAILURE;
    }

    std::ifstream file1(argv[1]);
    std::ifstream file2(argv[2]);
    std::ofstream outFile(argv[3]);

    if (!file1.is_open() || !file2.is_open() || !outFile.is_open()) {
        handleError("Failed to open one of the files");
    }

    std::string line1, line2;
    bool readFile1 = static_cast<bool>(std::getline(file1, line1));
    bool readFile2 = static_cast<bool>(std::getline(file2, line2));

    while (readFile1 || readFile2) {
        if (readFile1 && (!readFile2 || line1 <= line2)) {
            outFile << line1 << '\n';
            readFile1 = static_cast<bool>(std::getline(file1, line1));
        } else {
            outFile << line2 << '\n';
            readFile2 = static_cast<bool>(std::getline(file2, line2));
        }
    }

    file1.close();
    file2.close();
    outFile.close();
    return 0;
}
```

4. **Process Control and Inter-Process Communication:**

Create a Child Process with fork: Write a C++ program that uses fork to create a child process. The parent process should print "Parent Process", and the child process should print "Child Process". Use wait or similar system calls to ensure the parent waits for the child to finish before exiting.

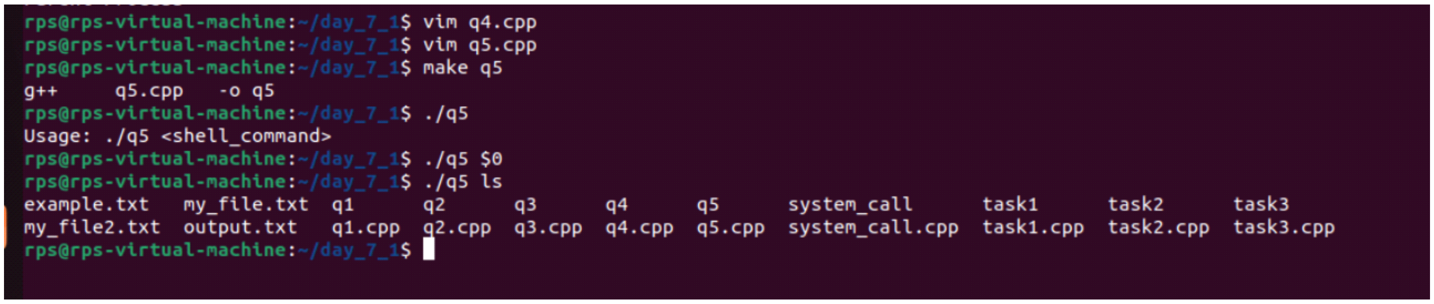


```
rps@rps-virtual-machine:~/day_7_1$ vim q4.cpp
rps@rps-virtual-machine:~/day_7_1$ make q4
g++     q4.cpp   -o q4
rps@rps-virtual-machine:~/day_7_1$ ./q4
Child Process
Parent Process
rps@rps-virtual-machine:~/day_7_1$
```

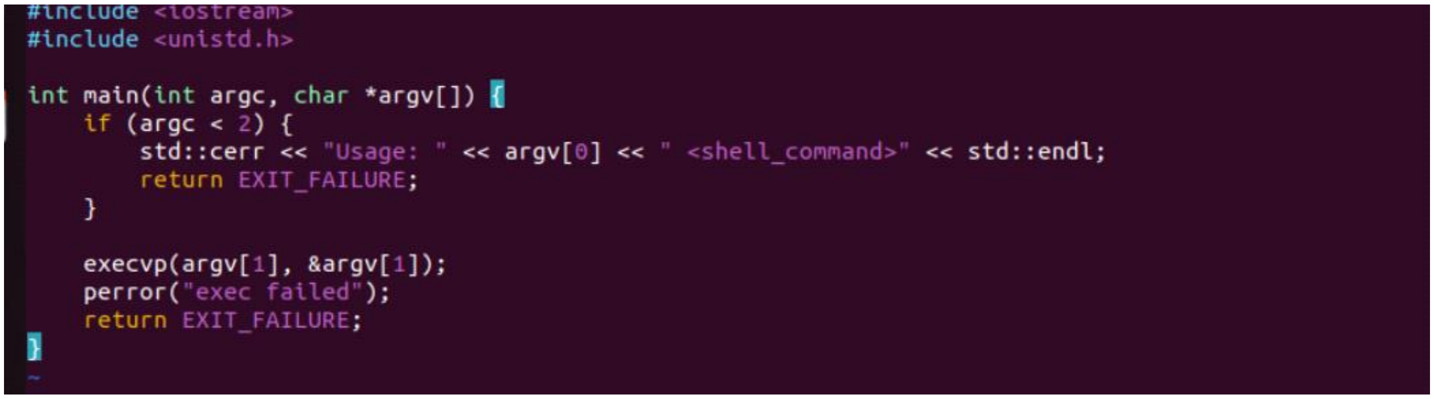


```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        return EXIT_FAILURE;
    } else if (pid == 0) {
        std::cout << "Child Process" << std::endl;
    } else {
        wait(NULL); // Wait for child process to finish
        std::cout << "Parent Process" << std::endl;
    }

    return 0;
}
~
```

5. **Execute a Shell Command: Write a C++ program that takes a shell command as a string argument and uses exec system calls (e.g., execlp or execv) to execute that command. Handle errors if the command execution fails.**

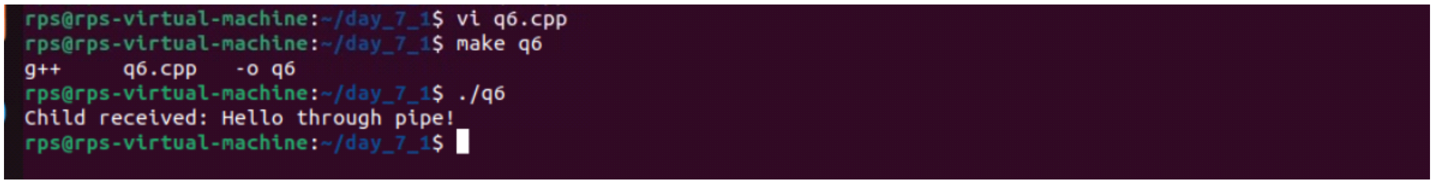


```
rps@rps-virtual-machine:~/day_7_1$ vim q4.cpp
rps@rps-virtual-machine:~/day_7_1$ vim q5.cpp
rps@rps-virtual-machine:~/day_7_1$ make q5
g++     q5.cpp   -o q5
rps@rps-virtual-machine:~/day_7_1$ ./q5
Usage: ./q5 <shell_command>
rps@rps-virtual-machine:~/day_7_1$ ./q5 $0
rps@rps-virtual-machine:~/day_7_1$ ./q5 ls
example.txt   my_file.txt  q1      q2      q3      q4      q5      system_call      task1      task2      task3
my_file2.txt  output.txt   q1.cpp  q2.cpp  q3.cpp  q4.cpp  q5.cpp  system_call.cpp  task1.cpp  task2.cpp  task3.cpp
rps@rps-virtual-machine:~/day_7_1$ 
```

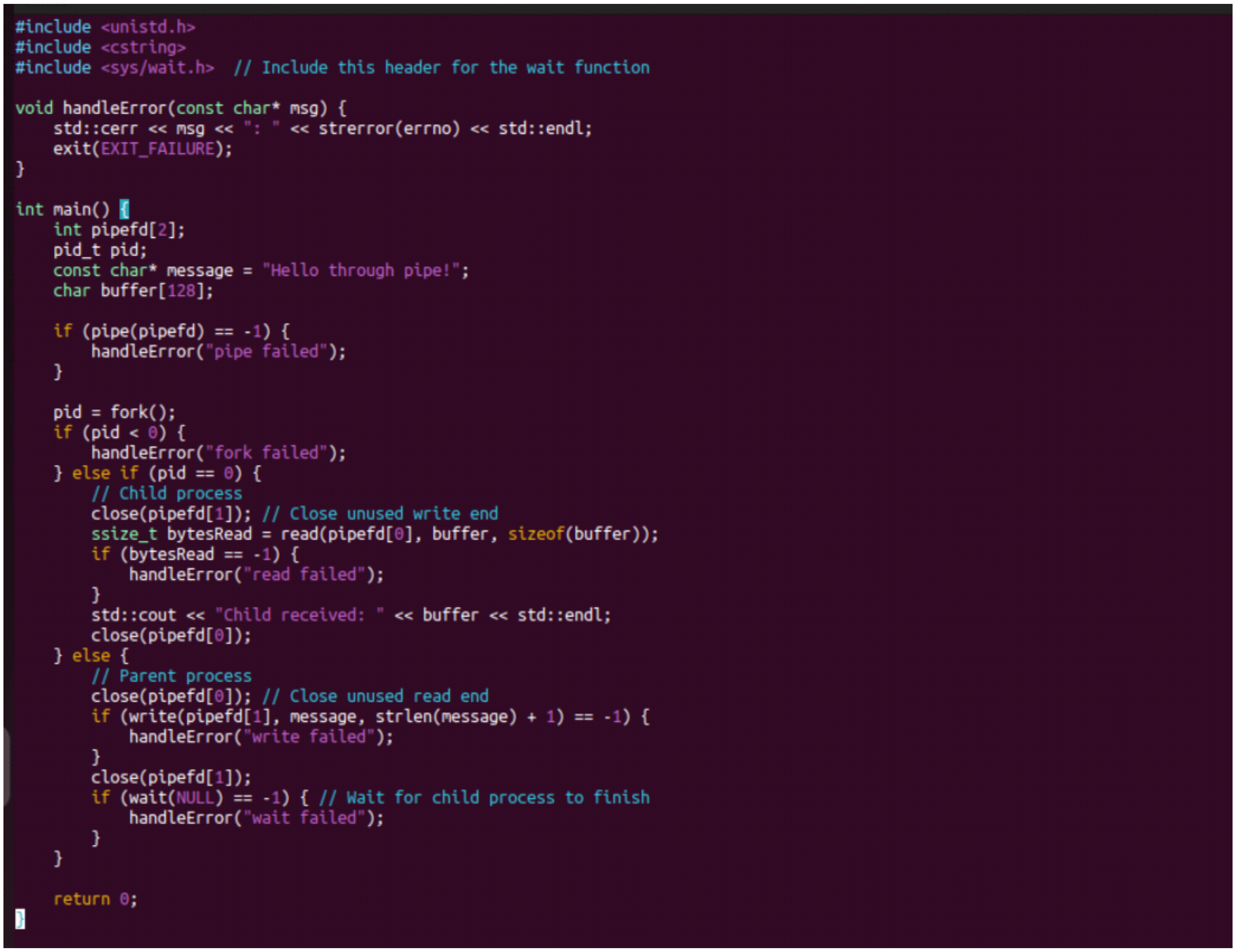


```
#include <iostream>
#include <unistd.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <shell_command>" << std::endl;
        return EXIT_FAILURE;
    }

    execvp(argv[1], &argv[1]);
    perror("exec failed");
    return EXIT_FAILURE;
}
~
```

6. Inter-Process Communication with Pipes: Write a C++ program that demonstrates inter-process communication using pipes. One process should write data to a pipe, and another process should read from the pipe and print the received data. Leverage pipe and fork system calls.

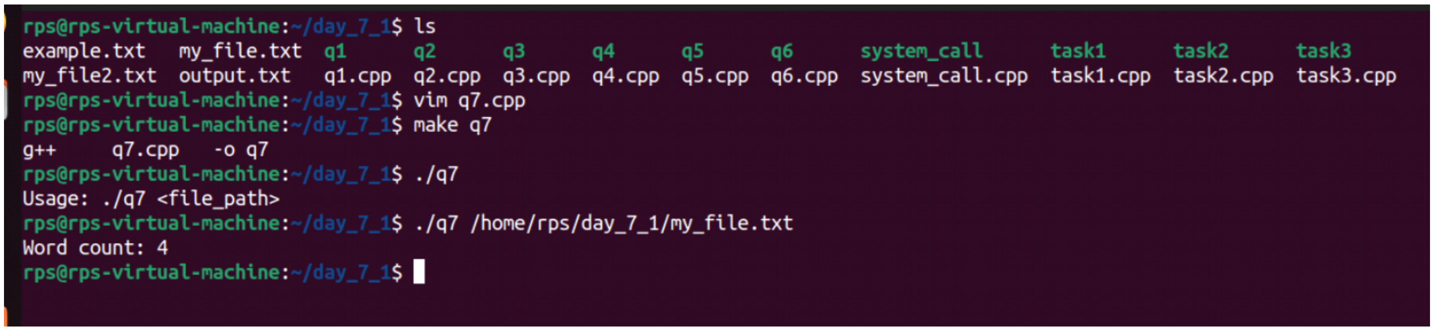


```
rps@rps-virtual-machine:~/day_7_1$ vi q6.cpp
rps@rps-virtual-machine:~/day_7_1$ make q6
g++     q6.cpp   -o q6
rps@rps-virtual-machine:~/day_7_1$ ./q6
Child received: Hello through pipe!
rps@rps-virtual-machine:~/day_7_1$ 
```

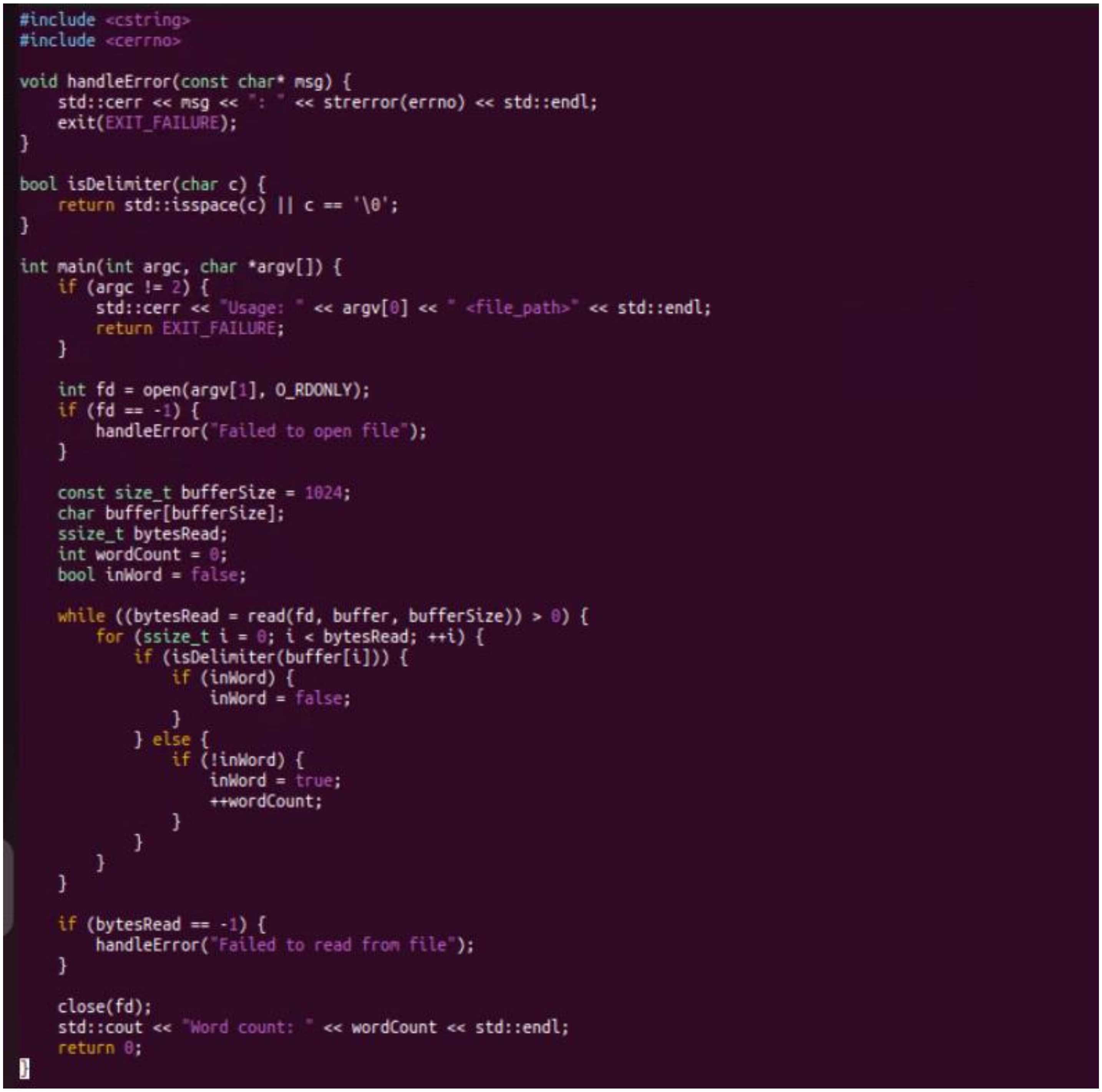

```
#include <unistd.h>
#include <cstring>
#include <sys/wait.h>  // Include this header for the wait function

void handleError(const char* msg) {
    std::cerr << msg << ": " << strerror(errno) << std::endl;
    exit(EXIT_FAILURE);
}

int main() {
    int pipefd[2];
    pid_t pid;
    const char* message = "Hello through pipe!";
    char buffer[128];

    if (pipe(pipefd) == -1) {
        handleError("pipe failed");
    }

    pid = fork();
    if (pid < 0) {
        handleError("fork failed");
    } else if (pid == 0) {
        // Child process
        close(pipefd[1]); // Close unused write end
        ssize_t bytesRead = read(pipefd[0], buffer, sizeof(buffer));
        if (bytesRead == -1) {
            handleError("read failed");
        }
        std::cout << "Child received: " << buffer << std::endl;
        close(pipefd[0]);
    } else {
        // Parent process
        close(pipefd[0]); // Close unused read end
        if (write(pipefd[1], message, strlen(message) + 1) == -1) {
            handleError("write failed");
        }
        close(pipefd[1]);
        if (wait(NULL) == -1) { // Wait for child process to finish
            handleError("wait failed");
        }
    }

    return 0;
}
```

Text Processing and System Information:

7.Count Words in a File: Write a C++ program that reads a text file and counts the number of words in it. Use open, read, and close system calls to access the file. Be mindful of delimiters and whitespace characters when counting words.

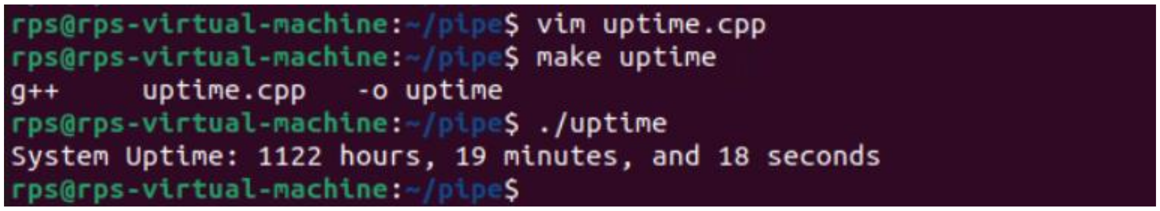


```
rps@rps-virtual-machine:~/day_7_1$ ls
example.txt   my_file.txt  q1      q2      q3      q4      q5      q6      system_call      task1      task2      task3
my_file2.txt  output.txt   q1.cpp  q2.cpp  q3.cpp  q4.cpp  q5.cpp  q6.cpp  system_call.cpp  task1.cpp  task2.cpp  task3.cpp
rps@rps-virtual-machine:~/day_7_1$ vim q7.cpp
rps@rps-virtual-machine:~/day_7_1$ make q7
g++     q7.cpp   -o q7
rps@rps-virtual-machine:~/day_7_1$ ./q7
Usage: ./q7 <file_path>
rps@rps-virtual-machine:~/day_7_1$ ./q7 /home/rps/day_7_1/my_file.txt
Word count: 4
rps@rps-virtual-machine:~/day_7_1$ 
```

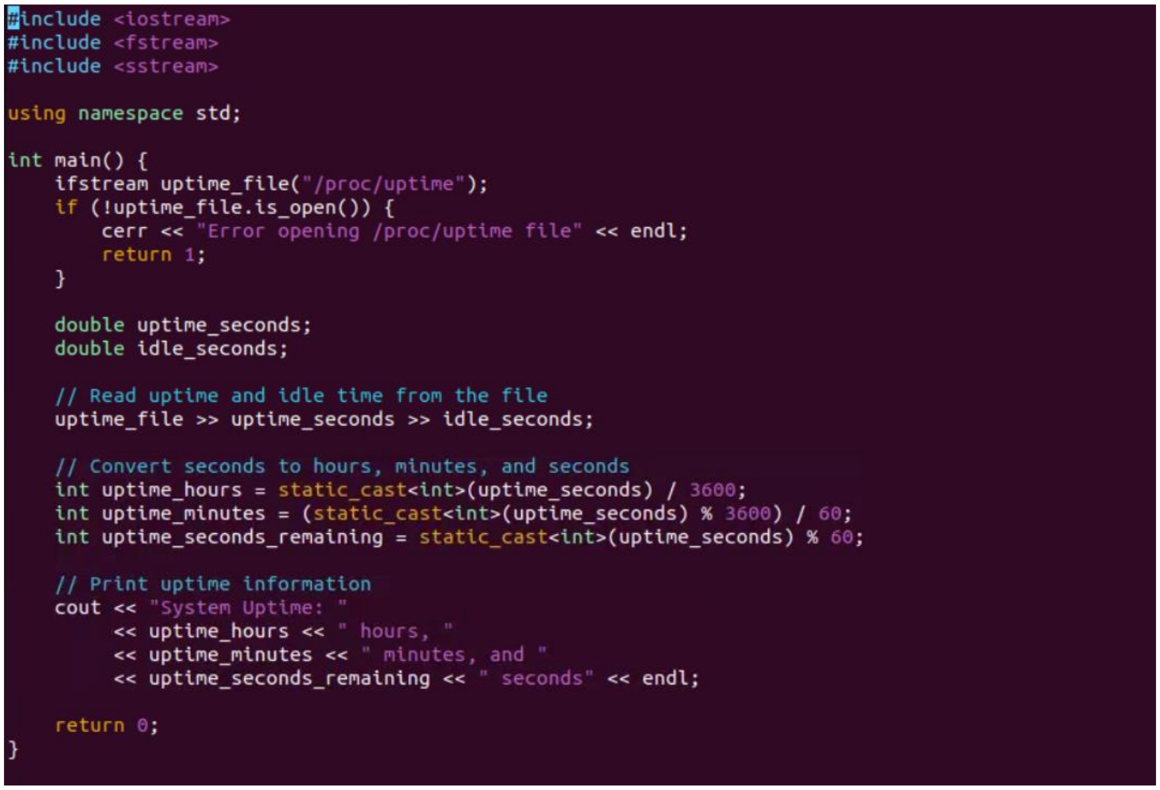


```
#include <cstring>
#include <cerrno>

void handleError(const char* msg) {
    std::cerr << msg << ": " << strerror(errno) << std::endl;
    exit(EXIT_FAILURE);
}

bool isDelimiter(char c) {
    return std::isspace(c) || c == '\0';
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        std::cerr << "Usage: " << argv[0] << " <file_path>" << std::endl;
        return EXIT_FAILURE;
    }

    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        handleError("Failed to open file");
    }

    const size_t bufferSize = 1024;
    char buffer[bufferSize];
    ssize_t bytesRead;
    int wordCount = 0;
    bool inWord = false;

    while ((bytesRead = read(fd, buffer, bufferSize)) > 0) {
        for (ssize_t i = 0; i < bytesRead; ++i) {
            if (isDelimiter(buffer[i])) {
                if (inWord) {
                    inWord = false;
                }
            } else {
                if (!inWord) {
                    inWord = true;
                    ++wordCount;
                }
            }
        }
    }

    if (bytesRead == -1) {
        handleError("Failed to read from file");
    }

    close(fd);
    std::cout << "Word count: " << wordCount << std::endl;
    return 0;
}
```

8. Get System Uptime: Write a C++ program that retrieves the system's uptime (time since it was last booted) using appropriate system calls (e.g., getuptime on Linux). Display the uptime information in a user-friendly format.



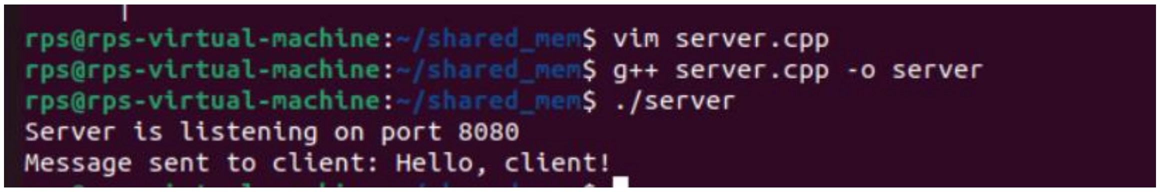
```
rps@rps-virtual-machine:~/pipe$ vim uptime.cpp
rps@rps-virtual-machine:~/pipe$ make uptime
g++     uptime.cpp   -o uptime
rps@rps-virtual-machine:~/pipe$ ./uptime
System Uptime: 1122 hours, 19 minutes, and 18 seconds
rps@rps-virtual-machine:~/pipe$
```



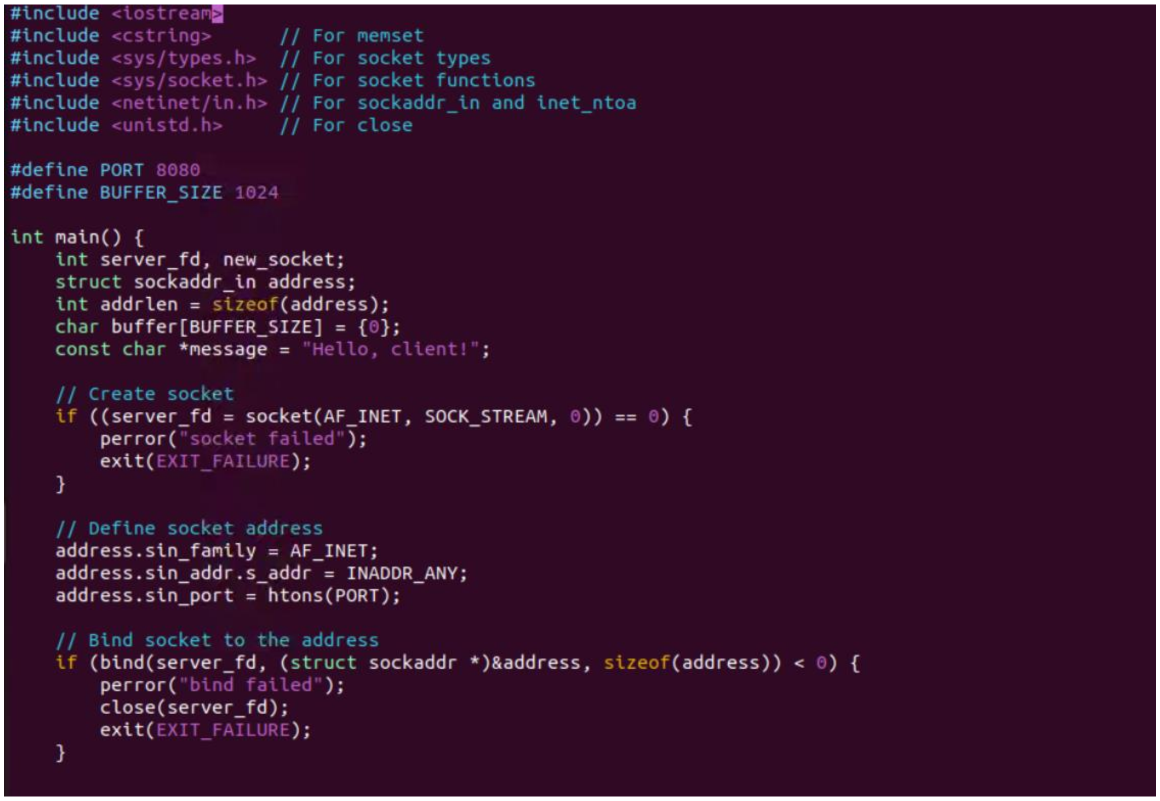
```cpp
#include <iostream>
#include <fstream>
#include <sstream>

using namespace std;

int main() {
    ifstream uptime_file("/proc/uptime");
    if (!uptime_file.is_open()) {
        cerr << "Error opening /proc/uptime file" << endl;
        return 1;
    }

    double uptime_seconds;
    double idle_seconds;

    // Read uptime and idle time from the file
    uptime_file >> uptime_seconds >> idle_seconds;

    // Convert seconds to hours, minutes, and seconds
    int uptime_hours = static_cast<int>(uptime_seconds) / 3600;
    int uptime_minutes = (static_cast<int>(uptime_seconds) % 3600) / 60;
    int uptime_seconds_remaining = static_cast<int>(uptime_seconds) % 60;

    // Print uptime information
    cout << "System Uptime: "
         << uptime_hours << " hours, "
         << uptime_minutes << " minutes, and "
         << uptime_seconds_remaining << " seconds" << endl;

    return 0;
}
```

Network Programming (Bonus):

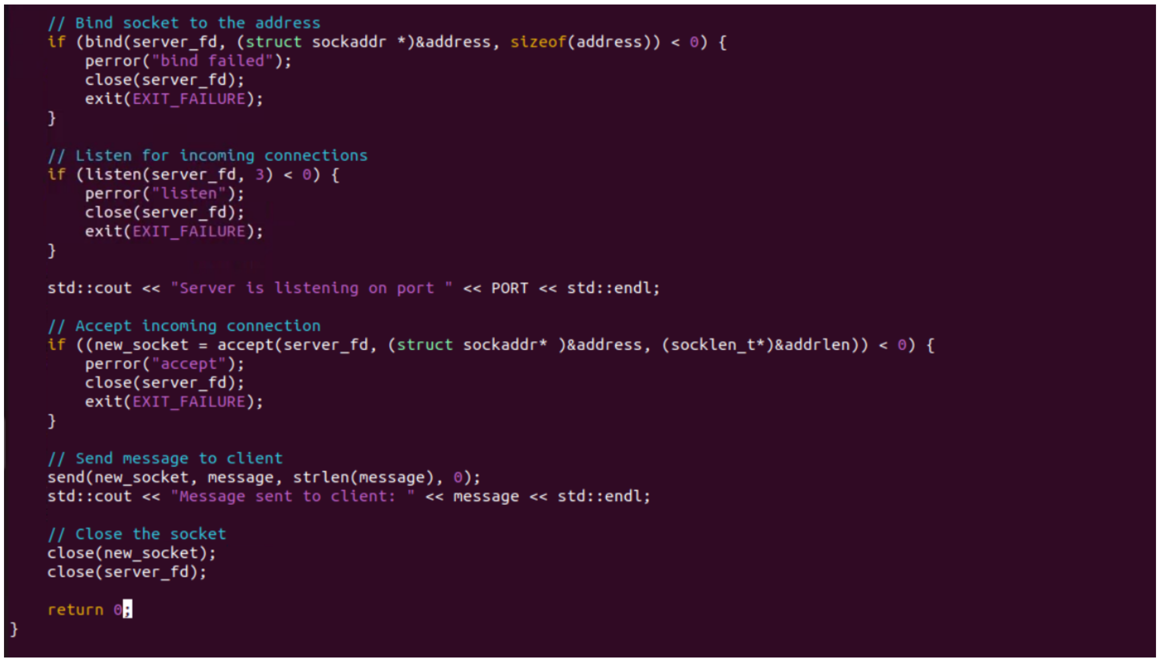
9. Simple TCP Server: Write a basic C++ program that acts as a server that listens for incoming TCP connections on a specific port. Upon receiving a connection, the server should send a predefined message (e.g., "Hello, client!") to the client and then close the connection. Utilize system calls like socket, bind, listen, accept, send, and recv. (Note: Network programming involves additional libraries/headers. Refer to system documentation)



```
rps@rps-virtual-machine:~/shared_mem$ vim server.cpp
rps@rps-virtual-machine:~/shared_mem$ g++ server.cpp -o server
rps@rps-virtual-machine:~/shared_mem$ ./server
Server is listening on port 8080
Message sent to client: Hello, client!
```



```cpp
#include <iostream>
#include <cstring>      // For memset
#include <sys/types.h>  // For socket types
#include <sys/socket.h> // For socket functions
#include <netinet/in.h> // For sockaddr_in and inet_ntoa
#include <unistd.h>     // For close

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[BUFFER_SIZE] = {0};
    const char *message = "Hello, client!";

    // Create socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Define socket address
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Bind socket to the address
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }
```

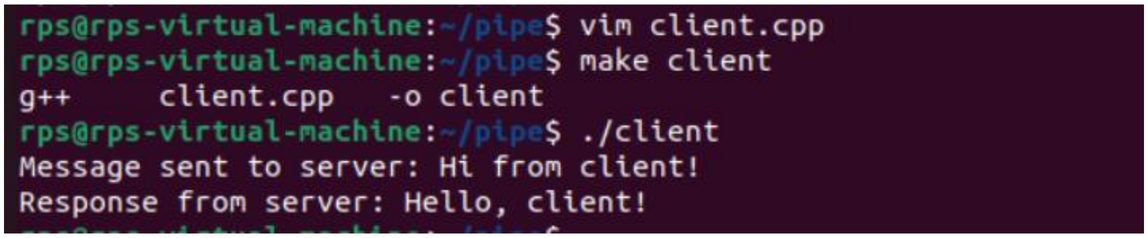


```cpp
    // Bind socket to the address
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }
```

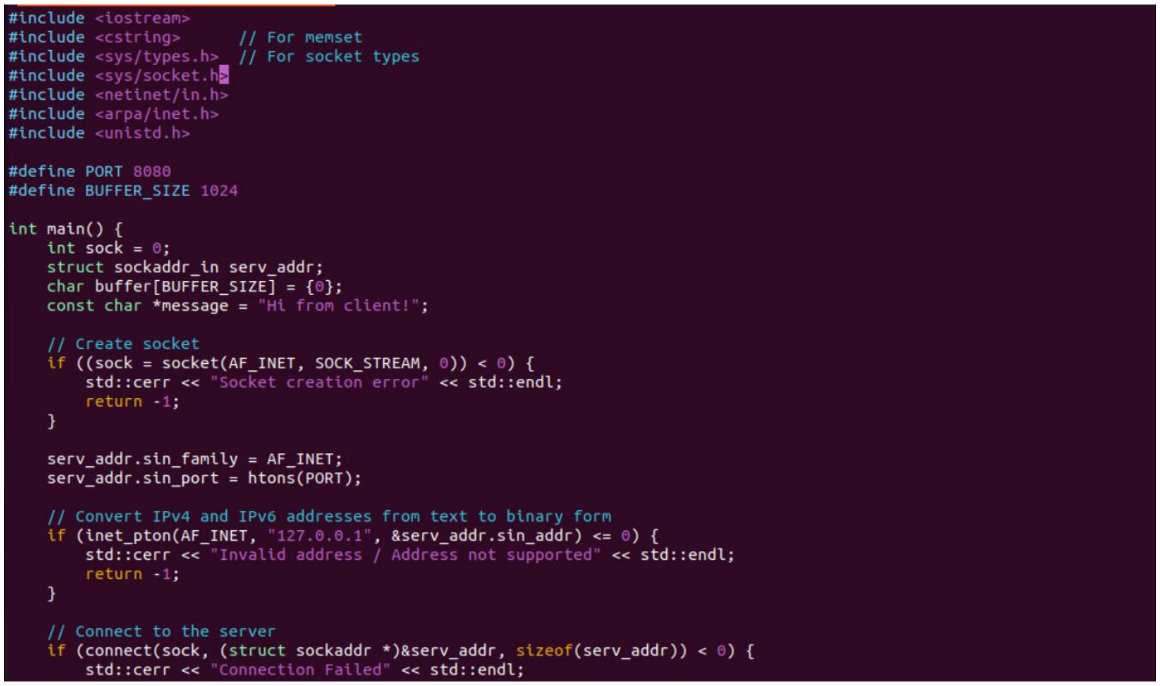
```cpp
    // Listen for incoming connections
    if (listen(server_fd, 3) < 0) {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    std::cout << "Server is listening on port " << PORT << std::endl;

    // Accept incoming connection
    if ((new_socket = accept(server_fd, (struct sockaddr* )&address, (socklen_t*)&addrlen)) < 0) {
        perror("accept");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // Send message to client
    send(new_socket, message, strlen(message), 0);
    std::cout << "Message sent to client: " << message << std::endl;

    // Close the socket
    close(new_socket);
    close(server_fd);

    return 0;
}
```

10. Simple TCP Client: As a companion to question 9, write a C++ program that acts as a client that connects to the server created in question 9. The client should send a message (e.g., "Hi from client!")

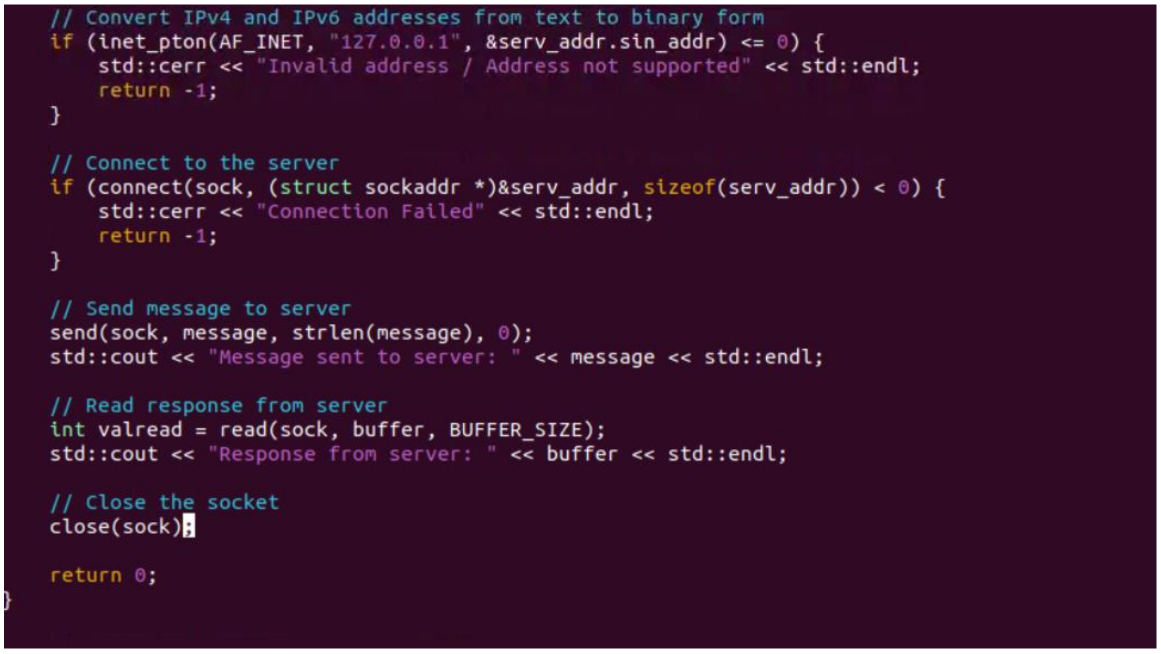
to the server, receive the server's response, and then close the connection. (Note: Network programming details apply here as well)



```
rps@rps-virtual-machine:~/pipe$ vim client.cpp
rps@rps-virtual-machine:~/pipe$ make client
g++     client.cpp   -o client
rps@rps-virtual-machine:~/pipe$ ./client
Message sent to server: Hi from client!
Response from server: Hello, client!
```



```
#include <iostream>
#include <cstring>      // For memset
#include <sys/types.h>  // For socket types
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};
    const char *message = "Hi from client!";

    // Create socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        std::cerr << "Socket creation error" << std::endl;
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        std::cerr << "Invalid address / Address not supported" << std::endl;
        return -1;
    }

    // Connect to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        std::cerr << "Connection Failed" << std::endl;
```



```
    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        std::cerr << "Invalid address / Address not supported" << std::endl;
        return -1;
    }

    // Connect to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        std::cerr << "Connection Failed" << std::endl;
        return -1;
    }

    // Send message to server
    send(sock, message, strlen(message), 0);
    std::cout << "Message sent to server: " << message << std::endl;

    // Read response from server
    int valread = read(sock, buffer, BUFFER_SIZE);
    std::cout << "Response from server: " << buffer << std::endl;

    // Close the socket
    close(sock);

    return 0;
}
```