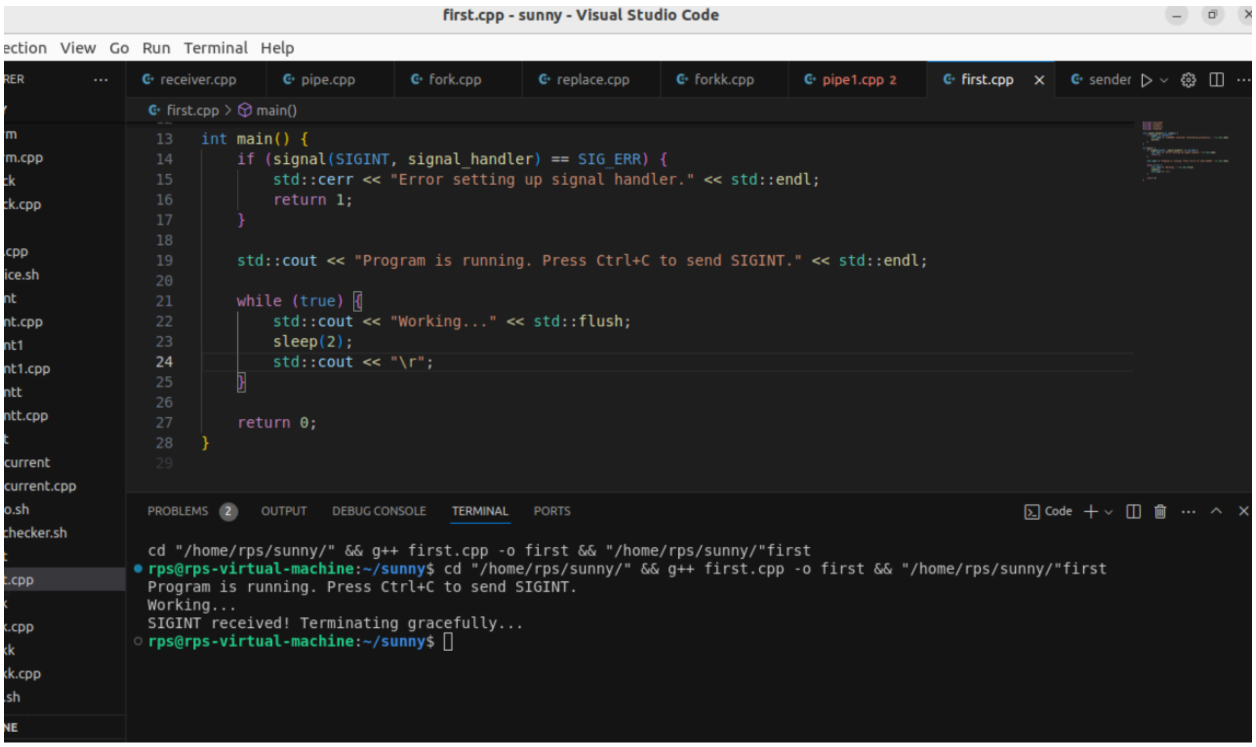
```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/msg.h>
#include <semaphore.h>
#include <errno.h>

#define PIPE1 "/tmp/pipe1"
#define PIPE2 "/tmp/pipe2"
#define SHM_SIZE 1024
#define MSG_KEY 1234
#define SEM_NAME "/my_sem"

// Define the message queue structure
struct MsgBuf {
    long mtype;
    char mtext[256];
};

// Semaphore functions
sem_t *create_semaphore(const char *name, int initial_value) {
    sem_t *sem = sem_open(name, O_CREAT | O_EXCL, 0666, initial_value);
    if (sem == SEM_FAILED) {
        perror("sem_open");
        exit(EXIT_FAILURE);
    }
    return sem;
}

void wait_semaphore(sem_t *sem) {
    if (sem_wait(sem) == -1) {
        perror("sem_wait");
        exit(EXIT_FAILURE);
    }
}

void post_semaphore(sem_t *sem) {
    if (sem_post(sem) == -1) {
        perror("sem_post");
        exit(EXIT_FAILURE);
    }
}

void close_semaphore(sem_t *sem) {
    if (sem_close(sem) == -1) {
        perror("sem_close");
        exit(EXIT_FAILURE);
    }
}

void unlink_semaphore(const char *name) {
    if (sem_unlink(name) == -1) {
        perror("sem_unlink");
        exit(EXIT_FAILURE);
    }
}

int main() {
    int pipe_fd1[2], pipe_fd2[2];
    int shm_id;
    char *shm_ptr;
    int msg_queue_id;
    sem_t *sem;

    // Create pipes
    if (pipe(pipe_fd1) == -1 || pipe(pipe_fd2) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }


    shm_id = shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT | 0666);
    if (shm_id == -1) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }
    shm_ptr = (char *)shmat(shm_id, NULL, 0);
    if (shm_ptr == (char *)-1) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
```

```
// Create message queue
msg_queue_id = msgget(MSG_KEY, IPC_CREAT | 0666);
if (msg_queue_id == -1) {
    perror("msgget");
    exit(EXIT_FAILURE);
}

// Create semaphore
sem = create_semaphore(SEM_NAME, 1);

pid_t pid = fork();
if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
}

if (pid == 0) { // Child process
    close(pipe_fd1[1]);
    close(pipe_fd2[0]);

    // Read from pipe
    char buffer[256];
    ssize_t bytes_read = read(pipe_fd1[0], buffer, sizeof(buffer));
    if (bytes_read == -1) {
        perror("read");
        exit(EXIT_FAILURE);
    }
    buffer[bytes_read] = '\0';
    std::cout << "Child received from pipe1: " << buffer << std::endl;

    // Write to shared memory
    wait_semaphore(sem);
    snprintf(shm_ptr, SHM_SIZE, "Hello from child!");
    post_semaphore(sem);

    // Send message
    MsgBuf msg;
    msg.mtype = 1;
    snprintf(msg.mtext, sizeof(msg.mtext), "Message from child");
    if (msgsnd(msg_queue_id, &msg, strlen(msg.mtext) + 1, 0) == -1) {
        perror("msgsnd");
        exit(EXIT_FAILURE);
    }
```

```
        close(pipe_fd1[0]);
        close(pipe_fd2[1]);
        exit(EXIT_SUCCESS);
    } else { // Parent process
        close(pipe_fd1[0]);
        close(pipe_fd2[1]);

        // Write to pipe
        const char *msg = "Hello from parent!";
        if (write(pipe_fd1[1], msg, strlen(msg) + 1) == -1) {
            perror("write");
            exit(EXIT_FAILURE);
        }

        // Wait for child to complete
        if (wait(NULL) == -1) {
            perror("wait");
            exit(EXIT_FAILURE);
        }

        // Read from shared memory
        wait_semaphore(sem);
        std::cout << "Parent read from shared memory: " << shm_ptr << std::endl;
        post_semaphore(sem);

        // Read message from message queue
        MsgBuf msg;
        if (msgrcv(msg_queue_id, &msg, sizeof(msg.mtext), 1, 0) == -1) {
            perror("msgrcv");
            exit(EXIT_FAILURE);
        }
        std::cout << "Parent received message: " << msg.mtext << std::endl;

        close(pipe_fd1[1]);
        close(pipe_fd2[0]);

        // Clean up
        if (shmdt(shm_ptr) == -1) {
            perror("shmdt");
            exit(EXIT_FAILURE);
        }
        if (shmctl(shm_id, IPC_RMID, NULL) == -1) {
            perror("shmctl");
            exit(EXIT_FAILURE);
        }
        if (msgctl(msg_queue_id, IPC_RMID, NULL) == -1) {
            perror("msgctl");
            exit(EXIT_FAILURE);
        }
        unlink_semaphore(SEM_NAME);
        close_semaphore(sem);
    }

    return 0;
}
```

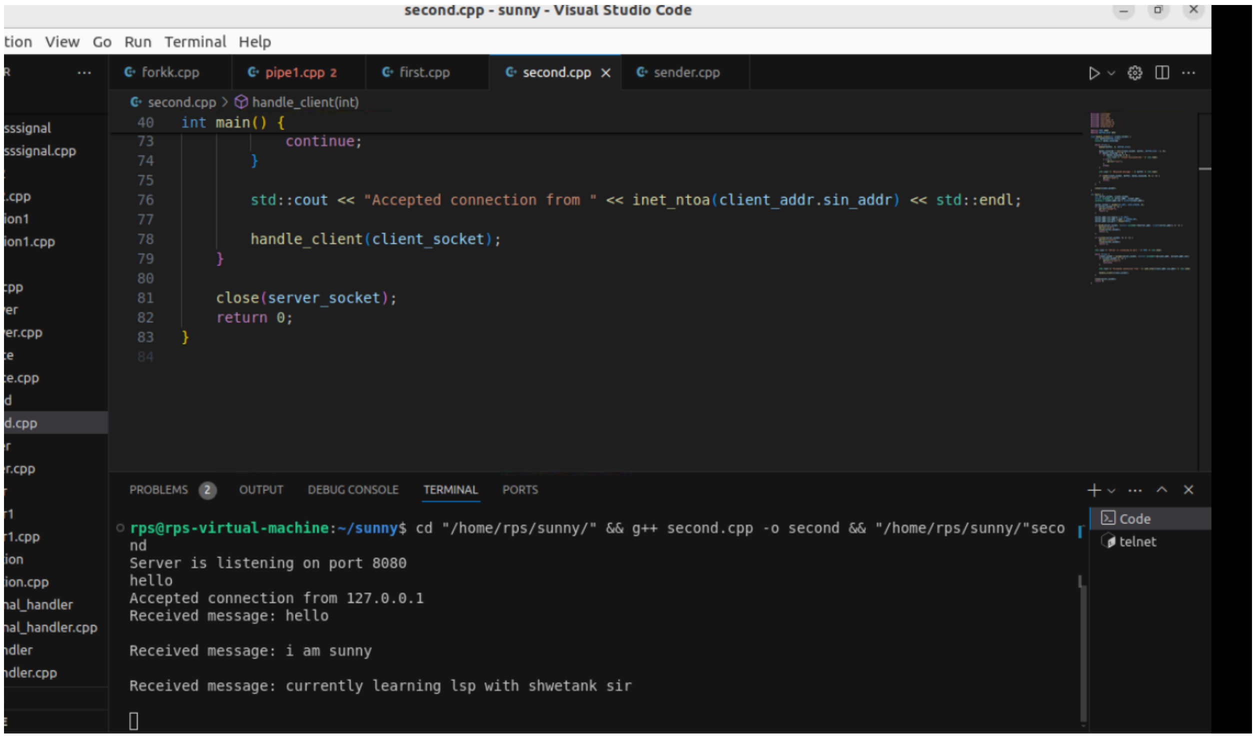
Q.Write a C++ program that sets up a signal handler for SIGINT. The program should perform some tasks and print a message when SIGINT is caught, then terminate gracefully.



```
int main() {
    if (signal(SIGINT, signal_handler) == SIG_ERR) {
        std::cerr << "Error setting up signal handler." << std::endl;
        return 1;
    }

    std::cout << "Program is running. Press Ctrl+C to send SIGINT." << std::endl;

    while (true) {
        std::cout << "Working..." << std::flush;
        sleep(2);
        std::cout << "\r";
    }

    return 0;
}
```

```
cd "/home/rps/sunny/" && g++ first.cpp -o first && "/home/rps/sunny/"first
rps@rps-virtual-machine:~/sunny$ cd "/home/rps/sunny/" && g++ first.cpp -o first && "/home/rps/sunny/"first
Program is running. Press Ctrl+C to send SIGINT.
Working...
SIGINT received! Terminating gracefully...
rps@rps-virtual-machine:~/sunny$ 
```

Q.Implement a simple echo server in C++ that listens on a specific port, accepts client connections, and echoes back any messages received from clients.



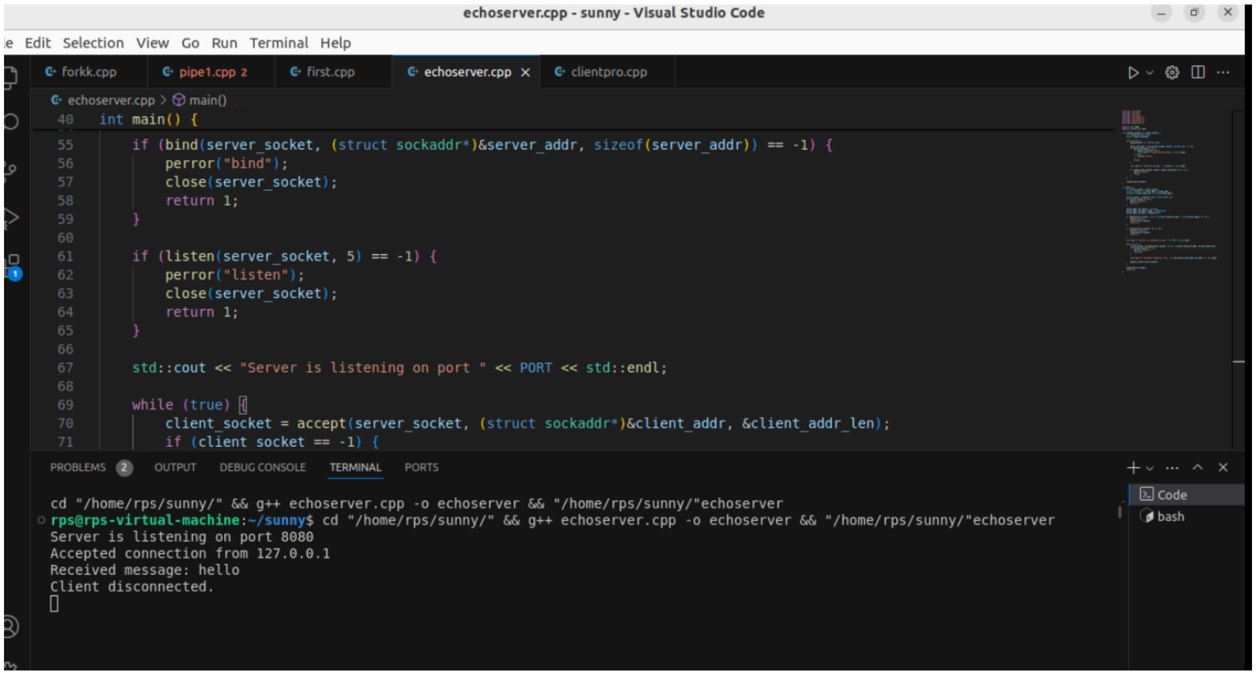
```
int main() {
            continue;
        }

        std::cout << "Accepted connection from " << inet_ntoa(client_addr.sin_addr) << std::endl;

        handle_client(client_socket);
    }

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

```
rps@rps-virtual-machine:~/sunny$ cd "/home/rps/sunny/" && g++ second.cpp -o second && "/home/rps/sunny/"seco
nd
Server is listening on port 8080
hello
Accepted connection from 127.0.0.1
Received message: hello

Received message: i am sunny

Received message: currently learning lsp with shwetank sir
```

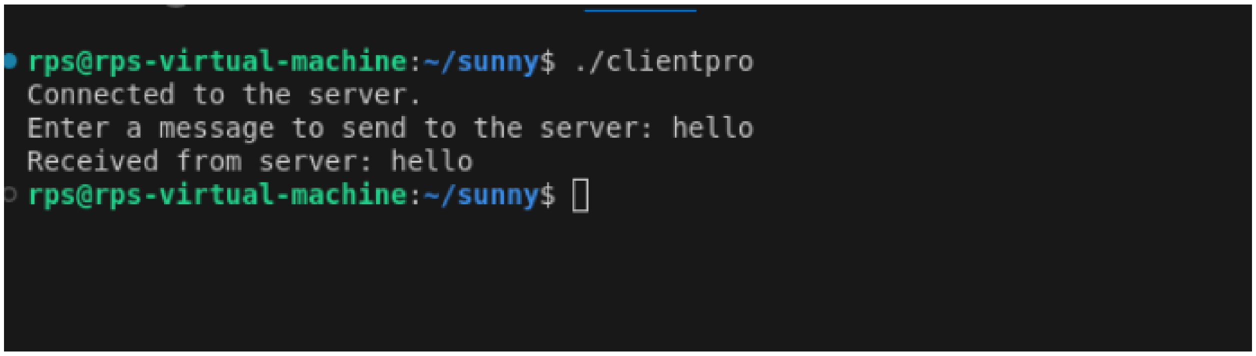
On the other terminal tab we need to run a command:-

Telnet localhost 8080 to receive and send message

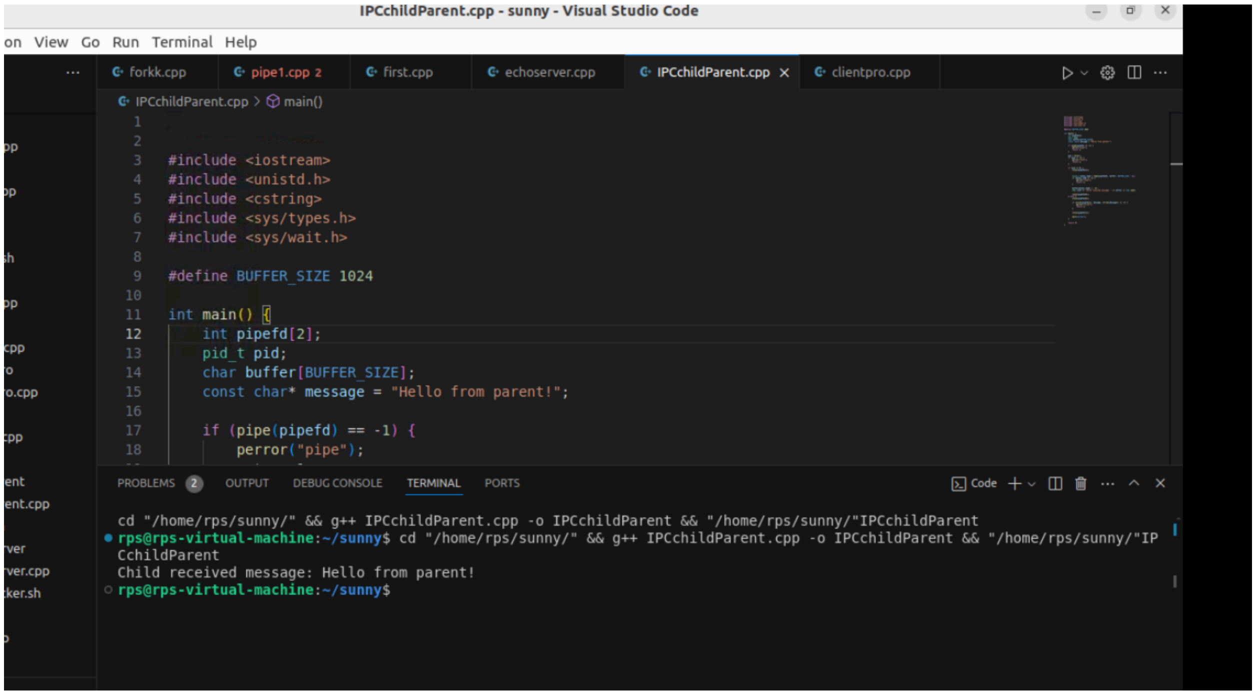
Q. Write a client program that connects to the echo server, sends a message, and prints the echoed response.


```
int main() {
    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind");
        close(server_socket);
        return 1;
    }

    if (listen(server_socket, 5) == -1) {
        perror("listen");
        close(server_socket);
        return 1;
    }

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

    while (true) {
        client_socket = accept(server_socket, (struct sockaddr*)&client_addr, &client_addr_len);
        if (client_socket == -1) {
```

```
cd "/home/rps/sunny/" && g++ echoserver.cpp -o echoserver && "/home/rps/sunny/"echoserver
rps@rps-virtual-machine:~/sunny$ cd "/home/rps/sunny/" && g++ echoserver.cpp -o echoserver && "/home/rps/sunny/"echoserver
Server is listening on port 8080
Accepted connection from 127.0.0.1
Received message: hello
Client disconnected.
```


```
rps@rps-virtual-machine:~/sunny$ ./clientpro
Connected to the server.
Enter a message to send to the server: hello
Received from server: hello
rps@rps-virtual-machine:~/sunny$
```

Q.Write a C++ program that creates a parent process and a child process. Use a pipe for IPC to send a message from the parent to the child, and have the child process print the message.



```cpp


#include <iostream>
#include <unistd.h>
#include <cstring>
#include <sys/types.h>
#include <sys/wait.h>

#define BUFFER_SIZE 1024

int main() {
    int pipefd[2];
    pid_t pid;
    char buffer[BUFFER_SIZE];
    const char* message = "Hello from parent!";

    if (pipe(pipefd) == -1) {
        perror("pipe");
```

```
cd "/home/rps/sunny/" && g++ IPCchildParent.cpp -o IPCchildParent && "/home/rps/sunny/"IPCchildParent
rps@rps-virtual-machine:~/sunny$ cd "/home/rps/sunny/" && g++ IPCchildParent.cpp -o IPCchildParent && "/home/rps/sunny/"IP
CchildParent
Child received message: Hello from parent!
rps@rps-virtual-machine:~/sunny$
```