# 1. shared.h

/\*

\* FILE: shared.h

\* PROGRAMMER: Tyler Gee, Cy Iver Torrefranca, Tuan Thanh Nguyen, George S.

\* PROJECT: SENG2031 - Assignment 1

\* DESCRIPTION:

\* shared.h contains shared definitions, structs, and function declarations

\* used by both the client and server programs.

\*/

#ifndef SHARED\_H

#define SHARED\_H

// FIFO definitions ----> Not sure which one to use for final copy

#define FIFO\_PATH "./travel\_agency\_fifo"

#define PERM\_OWNER\_RW 0600 // (Owner: rw, Group: --, Other: --)

#define PERM\_OWNER\_RW\_ALL\_R 0644 // (Owner: rw, Group: r-, Other: r-)

#define PERM\_ALL\_RW 0666 // (Owner: rw, Group: rw, Other: rw)

#define PERM\_ALL\_RWX 0777 // (Owner: rwx, Group: rwx, Other: rwx)

// General purpose defines

#define MAX\_BUFFER\_SIZE 256

// Regex patterns for input validation (\*: 0 or more, +: 1 or more)

#define REGEX\_NAME "^[A-Z][a-z]\* [A-Z][a-z]\*$" // Format: Firstname Lastname

#define REGEX\_NUMBER "^[0-9]+$" // Format: Numbers only, At least one digit

// Client field defines

#define MAX\_NAME\_LEN 50

#define MAX\_ADDRESS\_LEN 200

#define MAX\_DESTINATION\_LEN 200

#define MIN\_CLIENT\_AGE 18

#define MAX\_CLIENT\_AGE 125

#define MAX\_CLIENTS 100

#define MAX\_AGE\_STR\_LEN 8

// Constant return codes

#define SUCCESS 0

#define ERROR -1

// Constant for timeout duration (in seconds)

#define TIMEOUT\_DURATION 120

#define CANCEL\_TIMEOUT 0

#define TIMEOUT\_SUCCESSFUL 0

// Constants for buffer size

#define BUFFER\_SIZE\_OF\_ZERO 0

#define BUFFER\_SIZE\_OF\_ONE 1

#define BUFFER\_SIZE\_OF\_TWO 2

#define BUFFER\_SIZE\_OF\_THREE 3

#define BUFFER\_SIZE\_OF\_FOUR 4

// Constants for summary size

#define SUMMARY\_SIZE 512

// Define of Client struct

typedef struct Client {

char firstName[MAX\_NAME\_LEN];

char lastName[MAX\_NAME\_LEN];

int age;

char address[MAX\_ADDRESS\_LEN];

} Client;

// Define of Party struct

typedef struct Trip {

char destination[MAX\_DESTINATION\_LEN];

int numberOfClients;

Client clients[MAX\_CLIENTS]; // Change to dynamic array?

} Trip;

// Stream-based input gathering functions

int clearStream(FILE \*stream);

int getInputFromStream(FILE \*stream, char \*destination, size\_t bufSize, bool keepNewline);

// Validation Utility Functions

void printInputError(const char \*fieldName, int errorCode, size\_t bufSize);

bool isNullTerminated(const char \*buffer, size\_t bufSize);

bool stringMatchesRegex(const char \*string, size\_t bufSize, const char \*pattern);

#endif // SHARED\_H

# 2. Makefile

################################################################################  
# Program #  
################################################################################  
# Name of the client executable (without extension)  
CLIENT\_APPNAME = client  
# Name of the server executable (without extension)  
SERVER\_APPNAME = server  
  
################################################################################  
# File Structure Linux #  
################################################################################  
# Source Directory  
SRCDIR = src  
# Dependency Directory  
IDIR = inc  
# Object Directory  
OBJDIR = obj  
# Executable Directory  
EXECDIR = bin  
  
################################################################################  
# File Names Linux #  
################################################################################  
# := Means evaluate immediately not at time of use  
# Client Source files  
CLIENT\_SRC := $(SRCDIR)/client.c  
# Server Source files  
SERVER\_SRC := $(SRCDIR)/server.c  
# Client Object files  
CLIENT\_OBJ := $(OBJDIR)/client.o  
# Server Object files  
SERVER\_OBJ := $(OBJDIR)/server.o  
# Client Executable  
CLIENT\_EXEC := $(EXECDIR)/client  
# Server Executable  
SERVER\_EXEC := $(EXECDIR)/server  
LOG\_FILE := travel\_agency.log  
FIFO\_PIPE := travel\_agency\_fifo  
  
################################################################################  
# C Compiler Settings Linux #  
################################################################################  
# ?= Means default to if not set  
CC ?= cc  
CSTANDARD ?= -std=c17  
CFLAGS := -Wall -Wextra -Wpedantic -Werror $(CSTANDARD) -I$(IDIR)  
  
################################################################################  
# Linux Targets #  
################################################################################  
# Declare phony targets (not real files)  
.PHONY: all client server run-client run-server clean clean-log clean-FIFO distclean  
  
# Default target: build client and server, then run both  
all: client server  
  
# Build client executable and run it  
client: $(CLIENT\_EXEC)  
  
# Build server executable and run it  
server: $(SERVER\_EXEC)  
  
# Create /obj and /bin (mkdir -p flag: No error if exists)  
$(OBJDIR) $(EXECDIR):  
 @echo "Creating directory $@..."  
 @mkdir -p $@  
  
# Compile client.c -> obj/client.o (order-only prerequisite Ensures /obj exists)  
$(CLIENT\_OBJ): $(CLIENT\_SRC) | $(OBJDIR)  
 $(CC) $(CFLAGS) -c $(CLIENT\_SRC) -o $(CLIENT\_OBJ)  
  
# Compile server.c -> obj/server.o (order-only prerequisite Ensures /obj exists)  
$(SERVER\_OBJ): $(SERVER\_SRC) | $(OBJDIR)  
 $(CC) $(CFLAGS) -c $(SERVER\_SRC) -o $(SERVER\_OBJ)  
  
# Link client.o → bin/client (order-only prerequisite Ensures /bin exists)  
$(CLIENT\_EXEC): $(CLIENT\_OBJ) | $(EXECDIR)  
 $(CC) $(CFLAGS) $(CLIENT\_OBJ) -o $(CLIENT\_EXEC)  
  
# Link server.o → bin/server (order-only prerequisite Ensures /bin exists)  
$(SERVER\_EXEC): $(SERVER\_OBJ) | $(EXECDIR)  
 $(CC) $(CFLAGS) $(SERVER\_OBJ) -o $(SERVER\_EXEC)  
  
# Run the client program  
run-client: $(CLIENT\_EXEC)  
 @echo "Running client..."  
 @./$(CLIENT\_EXEC)  
  
# Run the server program  
run-server: $(SERVER\_EXEC)  
 @echo "Running server..."  
 @./$(SERVER\_EXEC)  
   
# Clean build artifacts  
clean:  
 @echo "Removing build artifacts..."  
 @rm -f $(OBJDIR)/\*.o $(CLIENT\_EXEC) $(SERVER\_EXEC) || true  
 @echo "Build artifacts removed successfully."  
  
# Clean log files  
clean-log:  
 @echo "Removing log file..."  
 @rm -f $(LOG\_FILE)  
 @echo "All log files removed successfully..."  
  
# Clean FIFO files  
clean-FIFO:  
 @echo "Removing FIFO pipt..."  
 @rm -f $(FIFO\_PIPE)  
 @echo "FIFO pipe removed successfully..."  
  
# Clean all generated files  
distclean: clean clean-log clean-FIFO  
 @echo "Removing build directories..."  
 @rm -rf $(OBJDIR) $(EXECDIR)  
 @echo "All build directories removed."

# 3. client.c

/\*

\* FILE: client.c

\* PROGRAMMER: Tyler Gee, Cy Iver Torrefranca, Tuan Thanh Nguyen, George S.

\* PROJECT: SENG2031 - Assignment 1

\* DESCRIPTION:

\* The client program collects trip and client data from the user,

\* then it validates the input, and sends it to the server via a FIFO.

\*/

// Include necessary header files for client.c functions and variables

#include <errno.h>

#include <fcntl.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

#include <signal.h>

#include <regex.h>

#include "shared.h"

// Conversion functions

bool convertToInt(const char \*buffer, int \*result);

// Validation Utility Functions

void printInputError(const char \*fieldName, int errorCode, size\_t bufSize);

bool isNullTerminated(const char \*buffer, size\_t bufSize);

bool stringMatchesRegex(const char \*string, size\_t bufSize, const char \*pattern);

// Trip and Client Input Functions

bool getInputFromClient(

const char \*label, char \*buffer, size\_t bufSize, char \*destination

);

bool getTripDestination(char \*destination);

void splitClientName(const char \*buffer, size\_t bufSize, char \*firstName, char \*lastName);

bool getClientName(char \*firstName, char \*lastName);

bool getClientAge(int \*age);

bool getClientAddress(char \*address);

char \*clientToString(const Client \*client);

// FIFO Stream Functions

int writestringToFIFO(const char \*fifoname, const char \*string, bool showConnectionMsg);

// Timeout functions

void timeout\_handler(int sig);

void reset\_timeout(void);

int main(void) {

char buffer[MAX\_BUFFER\_SIZE] = {0}; // Buffer for user input

int numberOfClients = 0; // Number of clients in current party

int err = 0; // Error code for input validation

Trip tripIfo = {0};

// Variables for client input validation

bool quitProgram = false;

bool partyStarted = false;

bool awaitingClientInput = false;

bool endOfClientList = false;

bool isValidDestination = false;

bool isValidName = false;

bool isValidAge = false;

bool isValidAddress = false;

printf("Travel Agency Client\n");

printf("Note: Please ensure the server is running before proceeding.\n");

// Set up timeout handler for inactivity

signal(SIGALRM, timeout\_handler);

alarm(TIMEOUT\_DURATION); // 2 minutes = 120 seconds

do {

// ---------- PARTY / STOP LOOP ----------

printf("\nEnter 'party' to start a new party or 'stop' to exit: ");

if ((err = getInputFromStream(stdin, buffer, MAX\_BUFFER\_SIZE, false)) != 0) {

printInputError("Start/stop input", err, MAX\_BUFFER\_SIZE);

continue;

}

// Reset timeout on user activity

reset\_timeout();

partyStarted = stringMatchesRegex(buffer, MAX\_BUFFER\_SIZE, "^party$");

quitProgram = stringMatchesRegex(buffer, MAX\_BUFFER\_SIZE, "^stop$");

if (!partyStarted && !quitProgram) {

printf("Input must be 'party' or 'stop'\n");

continue;

} else if (quitProgram) {

// Send stop command to server

if (writestringToFIFO(FIFO\_PATH, "stop", false) == -1) {

printf("Error: Failed to write stop command to FIFO\n");

}

break;

}

// i enable party input loop

// - cy

// ---------- Start FIFO Stream ----------

if ((mkfifo(FIFO\_PATH, PERM\_ALL\_RW)) == ERROR) {

// FIFO might already exist, which is okay

if (errno != EEXIST) {

printf("Could not create FIFO pipe.\n");

return ERROR;

}

}

printf("FIFO pipe ready.\n");

// Write 'party' to FIFO

if (writestringToFIFO(FIFO\_PATH, buffer, true) == ERROR) {

printf("Error: Failed to write to FIFO\n");

return ERROR;

}

// ---------- TRIP DATA INPUT BEGINS: DESTINATION ----------

do {

isValidDestination = getTripDestination(tripIfo.destination);

} while (!isValidDestination);

// Check if user wants to stop during destination input

if (stringMatchesRegex(tripIfo.destination, MAX\_DESTINATION\_LEN, "^stop$")) {

printf("Stopping the program...\n");

if (writestringToFIFO(FIFO\_PATH, "stop", false) == SUCCESS) {

printf("Sent stop command to server.\n");

}

break; // Exit the main loop

}

// Write destination to FIFO //

if (writestringToFIFO(FIFO\_PATH, tripIfo.destination, false) == ERROR) {

printf("Error: Failed to write to FIFO\n");

return ERROR;

}

numberOfClients = 0;

// ---------- CLIENT LOOP ----------

while (!endOfClientList) {

printf(

"\nEnter 'client' to add a client or 'end' to finish the "

"party: "

);

if ((err = getInputFromStream(stdin, buffer, MAX\_BUFFER\_SIZE, false)) != SUCCESS) {

printInputError("Client/end input", err, MAX\_BUFFER\_SIZE);

continue;

}

// Reset timeout on user activity

reset\_timeout();

endOfClientList = stringMatchesRegex(buffer, MAX\_BUFFER\_SIZE, "^end$");

awaitingClientInput = stringMatchesRegex(buffer, MAX\_BUFFER\_SIZE, "^client$");

if (!awaitingClientInput && !endOfClientList) {

printf("Input must be 'client' or 'end'\n");

continue;

} else if (endOfClientList) {

printf("Finished gathering clients for this party.\n");

// Write "end" signal to indicate party completion

// this is just to signal the server that the party is over - cy

if (writestringToFIFO(FIFO\_PATH, "END\_PARTY", false) == ERROR) {

printf("Error: Failed to write end signal to FIFO\n");

return ERROR;

}

break; // back to party/stop

}

// ---------- CLIENT DATA INPUT ----------

do {

isValidName = getClientName(

tripIfo.clients[numberOfClients].firstName,

tripIfo.clients[numberOfClients].lastName

);

} while (!isValidName);

do {

isValidAge = getClientAge(&(tripIfo.clients[numberOfClients].age));

} while (!isValidAge);

do {

isValidAddress

= getClientAddress(tripIfo.clients[numberOfClients].address);

} while (!isValidAddress);

// Display client information in formatted style

// Tuan Thanh Nguyen

printf("\n-----------------------------\n");

printf("Client %d\n", numberOfClients + 1);

printf("Name : %s %s\n",

tripIfo.clients[numberOfClients].firstName,

tripIfo.clients[numberOfClients].lastName);

printf("Age : %d\n", tripIfo.clients[numberOfClients].age);

printf("Address : %s\n", tripIfo.clients[numberOfClients].address);

printf("-----------------------------\n\n");

char \*clientString = clientToString(&tripIfo.clients[numberOfClients]);

// check if client string allocation failed

if (!clientString) {

printf("Error: Failed to allocate memory for client string\n");

return ERROR;

} else {

// Write client string to FIFO

if (writestringToFIFO(FIFO\_PATH, clientString, false) == ERROR) {

printf("Error: Failed to write to FIFO\n");

return ERROR;

}

// free client string

if (clientString) {

free(clientString);

clientString = NULL;

}

numberOfClients++;

} // end of client information input loop

}

// reset all loop control variables for next party

partyStarted = false;

awaitingClientInput = false;

endOfClientList = false;

numberOfClients = 0;

memset(&tripIfo, 0, sizeof(Trip));

} while (!quitProgram);

printf("Exiting the program...\n");

return SUCCESS;

}

/\*

\* FUNCTION: timeout\_handler

\* PROGRAMMER: Cy Iver Torrefranca

\* DESCRIPTION:

\* Signal handler for SIGALRM. Terminates the client after timeout.

\* PARAMETERS:

\* int sig - Signal number (SIGALRM)

\* RETURN:

\* n/a (exits program)

\*/

void timeout\_handler(int sig) {

(void)sig; // Suppress unused parameter warning

printf("\n\nClient timeout: No activity for 2 minutes. Terminating client...\n");

exit(TIMEOUT\_SUCCESSFUL);

}

/\*

\* FUNCTION: reset\_timeout

\* PROGRAMMER: Cy Iver Torrefranca

\* DESCRIPTION: Resets the alarm timer to 2 minutes from current time.

\* PARAMETERS: n/a

\* RETURN: n/a

\*/

void reset\_timeout(void) {

alarm(CANCEL\_TIMEOUT); // Cancel current alarm

alarm(TIMEOUT\_DURATION); // Reset to 2 minutes (120 seconds)

}

// #####################################################################################################################

// Shared header file Function Definitions (shared.h)

// #####################################################################################################################

/\* FUNCTION: clearStream

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Clears the input stream until a newline character or EOF is found and

\* returns the number of characters cleared. Newline characters and EOF are

\* \*\*NOT\*\* included in the returned count.

\*

\* PARAMETERS:

\* FILE \*stream: Pointer to the Stream to clear

\*

\* RETURN:

\* int: The number of characters cleared from the stream

\* (Excluding newline and EOF).

\*/

int clearStream(FILE \*stream) {

int currentChar = 0;

int count = 0;

while ((currentChar = fgetc(stream)) != '\n' && currentChar != EOF) {

count++;

}

return count;

}

/\* FUNCTION: getInputFromStream

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Reads a line of text from a stream into the destination buffer.

\* At most (bufSize - 1) characters are stored before truncation -

\* automatic null terminator is added.

\*

\* The newline character at the end of the input line can be kept depending on

\* the keepNewline flag:

\* - true: newline is preserved IF present

\* - false: newline is removed IF present

\*

\* If the input line exceeds (bufSize - 1) bytes, all extra characters

\* (excluding newline and EOF characters) are discarded. An error code of

\* ENOBUFS will be returned

\*

\* \*\*NOTE:\*\* newline and EOF characters left in the stream DO NOT COUNT! No

\* error code will be thrown if only those characters are left in the stream.

\*

\* PARAMETERS:

\* FILE \*stream: Stream to read from

\* char \*destination: Pointer to the buffer to store intput string in.

\* size\_t bufSize: Max Size of buffer \*\*including null terminator \*\*

\* bool keepNewline: Flag to keep the newline character at the end

\*

\* RETURN:

\* EOF(-1): End-of-file encountered before input. Usually means EOF

\* 0: Success

\* EIO(5): Generic I/O error if no specific errno is set.

\* EINVAL(22): Invalid arguments or empty input.

\* ENOBUFS(105): Input exceeded buffer size and was discarded.

\*/

int getInputFromStream(

FILE \*stream, char \*destination, size\_t bufSize, bool keepNewline

) {

// validate input parameters

if (!stream || !destination || bufSize < BUFFER\_SIZE\_OF\_TWO) {

return EINVAL; // Must be at least 1 char + null terminator

}

// Attempt to read input from the stream

if (fgets(destination, (int)bufSize, stream) == NULL) {

if (feof(stream)) {

return EOF; // end-of-file reached

}

return errno ? errno : EIO;

}

size\_t len = strlen(destination);

// Reject empty input or input that is just a newline

if (len == BUFFER\_SIZE\_OF\_ZERO || (len == BUFFER\_SIZE\_OF\_ONE && destination[0] == '\n')) {

return EINVAL; // empty input

}

bool bufferFull = (len == bufSize - BUFFER\_SIZE\_OF\_ONE);

bool lastCharIsNewline = (destination[len - BUFFER\_SIZE\_OF\_ONE] == '\n');

// Remove the newline if keepNewline flag is false

if (!keepNewline && lastCharIsNewline) {

destination[len - BUFFER\_SIZE\_OF\_ONE] = '\0';

len--; // adjust length for consistency

}

// If buffer was full and the last character was not a newline

if (bufferFull && !lastCharIsNewline) {

/\* clearStream discards any extra characters until a newline or EOF is

found. It returns the number of excess characters \*\*NOT\*\* including the

newline \*/

if (clearStream(stream) > 0) {

return ENOBUFS; // input was too long

}

}

return 0; // success

}

/\* FUNCTION: printInputError

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Print an error message based on the common error code received from stream

\* reading. Error codes are defined in <errno.h>. Response messages are

\* specific to the fieldName passed using the fieldName parameter.

\*

\* PARAMETERS:

\* const char \*fieldName: Label for field that the input failed validation from

\* int errorCode: Error Code received from stream reading

\* size\_t bufSize: Used to print Max buffer size in error message

\*

\* RETURN: None.

\*/

void printInputError(const char \*fieldName, int errorCode, size\_t bufSize) {

if (errorCode == EINVAL) { // Invalid arguments or empty input

printf(

"Invalid %s input - Input cannot be empty or only whitespace.\n", fieldName

);

} else if (errorCode == ENOBUFS) { // Input exceeded buffer size

printf(

"Invalid %s input - Exceeded maximum input length of %zu "

"characters.\n",

fieldName, bufSize - BUFFER\_SIZE\_OF\_ONE

);

} else { // Unknown error code

printf("An unknown error occurred while reading input.\n");

}

}

/\* FUNCTION: isNullTerminated

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Checks whether a buffer contains a null terminator ('\0') within the

\* first bufSize bytes.

\* \*\*NOTE:\*\* This function does not check for a null terminator in the

\* rest of the buffer that exceeds bufSize bytes.

\*

\* PARAMETERS:

\* const char \*buffer: Pointer to the buffer to check.

\* size\_t bufSize: Max number of bytes to examine in the buffer.

\* RETURN:

\* true: A null terminator was found within the first bufSize bytes.

\* false: No null terminator was found.

\*/

bool isNullTerminated(const char \*buffer, size\_t bufSize) {

if (!buffer || bufSize == BUFFER\_SIZE\_OF\_ZERO) { // Invalid input

return false;

}

/\* Iterate through the buffer and check for a null terminator

size\_t is used to ensure the loop doesn't go beyond the buffer's

boundaries. (e.g., bufSize > INT\_MAX)\*/

for (size\_t i = 0; i < bufSize; i++) {

if (buffer[i] == '\0') {

return true;

}

}

return false;

}

/\*

\* FUNCTION: stringMatchesRegex

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Using a regex pattern, check if the given string matches the pattern. String

\* is null-terminated and does not exceed bufSize bytes.

\* PARAMETERS:

\* const char \*string: The string to match against the pattern.

\* size\_t bufSize: Max size of the buffer in bytes, including the null

\* terminator.

\* const char \*pattern: The regex pattern to match against.

\* RETURN:

\* true: The string matches the pattern.

\* false: The string does not match the pattern.

\*/

bool stringMatchesRegex(const char \*string, size\_t bufSize, const char \*pattern) {

if (!string || !pattern || bufSize <= BUFFER\_SIZE\_OF\_ZERO) { // invalid input parameters

return false;

}

if (!isNullTerminated(string, bufSize)) {

return false;

}

size\_t length = strlen(string);

if (length == 0 || length > (bufSize - BUFFER\_SIZE\_OF\_ONE)) {

return false;

}

regex\_t regex;

// REG\_EXTENDED allows +, \*, ^, $, etc.

if (regcomp(&regex, pattern, REG\_EXTENDED) != SUCCESS) {

return false; // Regex could not be compiled.

}

// Execute the compiled pattern against the input string.

int result = regexec(&regex, string, 0, NULL, 0);

regfree(&regex); // Free the compiled regular expression.

return result == SUCCESS;

}

// #####################################################################################################################

// Client Specific Function Definitions

// #####################################################################################################################

/\*

\* FUNCTION: convertToInt

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Attempts to convert a string to an integer using strtol().

\* If conversion fails or if the string contains a non-numerical character is

\* found, the function returns false.

\* PARAMETERS:

\* const char \*buffer: String to convert.

\* int \*result: Pointer to integer to store the result.

\* RETURN:

\* true: Conversion successful and base 10 integer stored in result.

\* false: Conversion failed (not an valid integer or contains non-digits).

\*/

bool convertToInt(const char \*buffer, int \*result) {

if (!buffer || !result) {

return false; // Invalid input

}

const int MAX\_INT\_DIGITS = 11; // 32-bit int including sign

char bufCopy[MAX\_INT\_DIGITS + 1];

snprintf(bufCopy, sizeof(bufCopy), "%s", buffer); // create copy

// Attempt to convert using strtol()

errno = 0; // clear error flag

char \*numEndPtr = NULL; // success if endPtr points to '\0'

long convertedValue = strtol(bufCopy, &numEndPtr, 10);

// Check for errors

if (errno != 0 || \*numEndPtr != '\0') {

return false; // conversion failed

}

// Assign the converted value and typecast to int

\*result = (int)convertedValue;

return true;

}

/\*

\* FUNCTION: getInputFromClient

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Reads a line of text from the client's stdin input stream and stores it in

\* the destination buffer. If the input exceeds bufSize - 1 characters or only

\* a newline characters is received, an error message related to the label is

\* printed to the console and the function returns false. In the case of an

\* error, all characters in the buffer are discarded.

\*

\* PARAMETERS:

\* const char \*label: Used to prompt the user for input or contents of related

\* error messages.

\* char \*buffer: Buffer to store input, \*\*always\*\* set if successful.

\* size\_t bufSize: Max size of the buffer in bytes, including the null

\* terminator.

\* char \*destination: pointer to a buffer to store the input string in

\* \*\*Optional\*\* Set to NULL to avoid storing input in

\* \*destination pointer.

\* RETURN:

\* true: Input was successfully read and stored in buffer (and destination).

\* false: An error occurred or was found during input reading and processing.

\*/

bool getInputFromClient(

const char \*label, char \*buffer, size\_t bufSize, char \*destination

) {

int err = 0;

printf("%s: ", label);

if ((err = getInputFromStream(stdin, buffer, bufSize, false)) == SUCCESS) {

// Reset timeout on successful input

reset\_timeout();

if (destination) {

snprintf(destination, bufSize, "%s", buffer);

}

return true;

}

printInputError(label, err, bufSize);

return false;

}

/\*

\* FUNCTION: getTripDestination

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Reads a line of text from the client's stdin input stream and stores it in

\* the destination buffer. If input exceeds MAX\_DESTINATION\_LEN - 1 characters,

\* an error message related to the label is printed to the console and the

\* function returns false.

\*

\* Use this function to set the destination field of a Trip struct.

\*

\* PARAMETERS:

\* char \*destination: Buffer to store the destination input in.

\*

\* RETURN:

\* true: Input was successfully read and stored in \*destination.

\* false: An error occurred or was found during input reading and processing.

\*/

bool getTripDestination(char \*destination) {

char buffer[MAX\_DESTINATION\_LEN] = {0};

return getInputFromClient("Destination", buffer, MAX\_DESTINATION\_LEN, destination);

}

/\* FUNCTION: splitClientName

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Split a client's full name into first and last names at the first

\* occurrence of a space.

\*

\* All characters before the space are copied in the buffer pointed to by

\* \*firstName, and all characters after the space are copied into the buffer

\* pointed to by \*lastName.

\*

\* IMPORTANT NOTE: Validation of input is not performed in this function, the

\* caller is responsible for ensuring the input is valid and safe:

\* - buffer is not NULL

\* - buffer contains at least one space.

\* - The buffer is null-terminated.

\* - firstName and lastName have enough space for copied values

\*

\* PARAMETERS:

\* const char \*buffer: buffer containing a space to split at

\* size\_t bufSize: Max size of the buffer in bytes, including \0

\* char \*firstName: Pointer to store the first split string into

\* char \*lastName Pointer to store the remaining bytes into

\* RETURN: None

\*/

void splitClientName(

const char \*buffer, size\_t bufSize, char \*firstName, char \*lastName

) {

char \*spacePosition = strchr(buffer, ' ');

if (spacePosition == NULL) { // Defensive check — should not happen

firstName[0] = '\0';

lastName[0] = '\0';

return;

}

int firstNameLength = spacePosition - buffer;

// -1 to exclude the space when calculating lastNameLength

// int lastNameLength = (int)(strlen(buffer) - firstNameLength - 1);

// Copy first name

// "%.\*s" prints N chars -> N = firstNameLength, snprintf() null terminates

// added +1 to firstNameLength to account for null terminator (missing letters) -cy

//

snprintf(firstName, firstNameLength + 1, "%.\*s", firstNameLength, buffer);

// Copy last name

snprintf(lastName, bufSize - firstNameLength - 1, "%s", spacePosition + 1);

}

/\*

\* FUNCTION: getClientName

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Prompts the user for a client's full name and validate input against the

\* regex pattern "REGEX\_NAME" and a max length of MAX\_CLIENT\_NAME\_LEN defined

\* in shared.h. If the input is valid, the full name input is split into first

\* and last names and stored in the provided buffers. If the input is invalid,

\* an error message is printed to the console and the function returns false.

\*

\* PARAMETERS:

\* char \*firstName: Buffer to store the first name into

\* char \*lastName: Buffer to store the last name into

\*

\* RETURN:

\* true: Input was successfully validated and split into first and last names.

\* false: An error occurred/was found during input validation and/or splitting.

\*/

bool getClientName(char \*firstName, char \*lastName) {

char buffer[MAX\_NAME\_LEN] = {0};

if (!getInputFromClient("Name", buffer, MAX\_NAME\_LEN, NULL)) {

return false;

}

if (!stringMatchesRegex(buffer, MAX\_NAME\_LEN, REGEX\_NAME)) {

printf("Invalid Name Input: Must be in 'First Last' format\n");

return false;

}

splitClientName(buffer, MAX\_NAME\_LEN, firstName, lastName);

return true;

}

/\*

\* FUNCTION: getClientAge

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Prompts the user for a client's age and validate input against the regex

\* pattern "REGEX\_NUMBER" defined in shared.h. If the input is valid attempt to

\* convert the input to an integer and store it in the provided pointer. If the

\* input is invalid, an error message is printed to the console and the

\* function returns false.

\*

\* If the input is outside the valid range MIN\_CLIENT\_AGE to MAX\_CLIENT\_AGE

\* (Provided in shared.h), an error message is printed to the console and the

\* function returns false.

\*

\* PARAMETERS:

\* int \*age: Pointer to store the client's age into

\*

\* RETURN:

\* true: Input was successfully validated and converted to an integer.

\* false: An error occurred/was found during input validation and/or conversion

\*/

bool getClientAge(int \*age) {

const size\_t bufSize = BUFFER\_SIZE\_OF\_FOUR; // Max age is 3 characters + null terminator

char buffer[bufSize];

// age cannot be NULL

if (!age) {

return false;

}

if (!getInputFromClient("Age", buffer, bufSize, NULL)) {

return false;

}

int result = 0;

bool isMatch = stringMatchesRegex(buffer, bufSize, REGEX\_NUMBER);

if (!isMatch || !convertToInt(buffer, &result)

|| (result < MIN\_CLIENT\_AGE || result > MAX\_CLIENT\_AGE)) {

printf(

"Invalid Age Input: Must be a number between %d and %d.\n", MIN\_CLIENT\_AGE,

MAX\_CLIENT\_AGE

);

return false;

}

\*age = result;

return true;

}

/\*

\* FUNCTION: getClientAddress

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Prompts the user for a client's address and store it in the provided buffer.

\* If input exceeds MAX\_ADDRESS\_LEN - 1 characters, an error message related to

\* the label is printed to the console and the function returns false.

\*

\* PARAMETERS:

\* char \*address: Buffer to store the client's address into

\*

\* RETURN:

\* true: Input was successfully read and stored in \*address.

\* false: An error occurred or was found during input reading and processing.

\*/

bool getClientAddress(char \*address) {

char buffer[MAX\_ADDRESS\_LEN] = {0};

return getInputFromClient("Address", buffer, MAX\_ADDRESS\_LEN, address);

}

/\*

\* FUNCTION: clientToString

\* PROGRAMMER: Tyler Gee

\* DESCRIPTION:

\* Converts a Client struct into a comma-separated string in the format:

\* "firstName,lastName,age,address". Dynamically allocates memory for

\* the returned string or returns NULL if any of the required fields are

\* missing, invalid or a memory allocation error occurs.

\*

\* The returned string must be freed by the caller.

\*

\* PARAMETERS:

\* const Client \*client: Pointer to the Client struct to convert to a string.

\*

\* RETURN:

\* char \*: Dynamically allocated string representing the client's information

\* int the format specified or NULL if an error occurred, a field is

\* missing or memory allocation fails.

\*

\*/

char \*clientToString(const Client \*client) {

if (!client || strlen(client->firstName) == BUFFER\_SIZE\_OF\_ZERO || strlen(client->lastName) == BUFFER\_SIZE\_OF\_ZERO

|| strlen(client->address) == BUFFER\_SIZE\_OF\_ZERO || client->age <= BUFFER\_SIZE\_OF\_ZERO) {

return NULL;

}

// (\*\_\_stream = NULL, \* \_\_n = 0) -> snprint calculates characters count

const int totalLength = snprintf(

NULL, 0, "%s,%s,%d,%s", client->firstName, client->lastName, client->age,

client->address

);

// Allocate memory for the client string + null terminator

char \*clientString = calloc(totalLength + 1, sizeof(char));

if (!clientString) {

printf("Memory allocation failed\n");

return NULL;

}

// Construct the client string

snprintf(

clientString, totalLength + 1, "%s,%s,%d,%s", client->firstName, client->lastName,

client->age, client->address

);

return clientString;

}

/\*

\* FUNCTION: writestringToFIFO

\* PROGRAMMER: Tyler Gee & Cy Iver Torrefranca

\* DESCRIPTION:

\* Writes a string to a named FIFO stream.

\* PARAMETERS:

\* const char \*fifoname: Name of the FIFO to write to.

\* const char \*string: String to write to the FIFO.

\* bool showConnectionMsg: If true, displays "Waiting for server..." and "Connected to server!" messages.

\* RETURN:

\* int: Returns 0 on success, ERROR on failure.

\*/

int writestringToFIFO(const char \*fifoname, const char \*string, bool showConnectionMsg) {

// Open FIFO stream for writing

if (showConnectionMsg) {

printf("Waiting for server...\n");

}

int fd = open(fifoname, O\_WRONLY);

if (fd == -1) { // Check for error

perror("Error opening FIFO stream for writing");

return ERROR;

}

if (showConnectionMsg) {

printf("Connected to server!\n");

}

// Write string plus newline to FIFO

size\_t len = strlen(string);

char \*buffer = malloc(len + 2); // +1 for newline, +1 for null terminator

if (!buffer) {

perror("Memory allocation failed");

close(fd);

return ERROR;

}

snprintf(buffer, len + 2, "%s\n", string);

ssize\_t bytesWritten = write(fd, buffer, len + 1);

if (bytesWritten == ERROR) { // Check for error

perror("Error writing to FIFO stream");

free(buffer);

close(fd);

return ERROR;

}

printf("Sent to server: %s\n", string);

free(buffer);

close(fd); // close fifo

return SUCCESS; // success

}

# 4. server.c

/\*

\* FILE: server.c

\* PROGRAMMER: Tyler Gee, Cy Iver Torrefranca, Tuan Thanh Nguyen, George S.

\* PROJECT: SENG2031 - Assignment 1

\* DESCRIPTION:

\* The server program receives trip and client data from the client via a FIFO,

\* processes the data, and logs the activities.

\*/

#include <fcntl.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/stat.h>

#include <time.h>

#include <unistd.h>

#include <errno.h>

#include <signal.h>

#include "shared.h"

void processMessages(const char \*fifoname);

void writeToLog(FILE \*logFile, const char \*message);

void timeout\_handler(int sig);

void reset\_timeout(void);

int main(void) {

printf("Travel Agency Server - Waiting for client data...\n");

// Set up timeout handler for inactivity

signal(SIGALRM, timeout\_handler);

alarm(TIMEOUT\_DURATION); // 2 minutes = 120 seconds

// Create FIFO if it doesn't exist

if (mkfifo(FIFO\_PATH, PERM\_OWNER\_RW\_ALL\_R) == -1) {

// FIFO might already exist, which is okay

if (errno != EEXIST) {

perror("Error creating FIFO");

return -1;

}

}

// Process messages from clients

processMessages(FIFO\_PATH);

return 0;

}

/\*

\* FUNCTION: processMessages

\* PROGRAMMER: Cy Iver Torrefranca & Tuan Thanh Nguyen

\* DESCRIPTION:

\* Processes messages from the FIFO, handling party and client data,

\* and logging activities to a log file.

\* PARAMETERS:

\* const char \*fifoname : Path to the FIFO to read messages from.

\* RETURNS : n/a

\*/

void processMessages(const char \*fifoname) {

FILE \*logFile = fopen("travel\_agency.log", "a");

if (!logFile) {

perror("Error opening log file");

return;

}

int fd;

char buffer[MAX\_BUFFER\_SIZE];

char destination[MAX\_DESTINATION\_LEN] = {0};

int clientCount = 0;

bool inParty = false;

bool serverRunning = true;

writeToLog(logFile, "Server started");

while (serverRunning) {

// Open FIFO for reading

/\*

O\_RDONLY is a flag that opens the FIFO in read-only mode.

If the FIFO does not exist or cannot be opened, open() returns -1 and sets errno.

source: https://pubs.opengroup.org/onlinepubs/7908799/xsh/open.html

\*/

fd = open(fifoname, O\_RDONLY);

if (fd == -1) {

perror("Error opening FIFO for reading");

break;

}

printf("Client connected.\n");

// Read message from FIFO

ssize\_t bytesRead = read(fd, buffer, sizeof(buffer) - 1);

if (bytesRead > 0) {

// Reset timeout on activity

reset\_timeout();

buffer[bytesRead] = '\0';

// Remove trailing newline if present

size\_t len = strlen(buffer);

if (len > 0 && buffer[len-1] == '\n') {

buffer[len-1] = '\0';

}

printf("Received: %s\n", buffer);

writeToLog(logFile, buffer);

// Process the message based on content

if (strcmp(buffer, "party") == SUCCESS) {

inParty = true;

clientCount = 0;

memset(destination, 0, sizeof(destination));

printf("New party started\n");

}

else if (strcmp(buffer, "stop") == SUCCESS) {

printf("Stop command received. Shutting down server.\n");

writeToLog(logFile, "Server received stop command");

serverRunning = false;

}

else if (inParty && strlen(destination) == SUCCESS) {

// First message after "party" should be destination

strncpy(destination, buffer, sizeof(destination) - 1);

printf("Party destination: %s\n", destination);

}

else if (strcmp(buffer, "client") == SUCCESS) {

printf("New client being added...\n");

}

else if (strcmp(buffer, "END\_PARTY") == SUCCESS || strcmp(buffer, "end") == SUCCESS) {

if (inParty) {

printf("=== PARTY SUMMARY ===\n");

printf("Destination: %s\n", destination);

printf("Number of clients: %d\n", clientCount);

printf("====================\n\n");

char summary[SUMMARY\_SIZE]; // Tuan Thanh Nguyen

snprintf(summary, sizeof(summary), "Party completed - Destination: %s, Clients: %d",

destination, clientCount);

writeToLog(logFile, summary);

}

inParty = false;

}

else if (inParty && strchr(buffer, ',') != NULL) {

// This looks like client data (contains commas)

clientCount++;

// Parse client data: "FirstName,LastName,Age,Address"

char firstName[MAX\_NAME\_LEN] = {0};

char lastName[MAX\_NAME\_LEN] = {0};

char ageStr[MAX\_AGE\_STR\_LEN] = {0};

char address[MAX\_ADDRESS\_LEN] = {0};

char temp[MAX\_BUFFER\_SIZE];

strncpy(temp, buffer, sizeof(temp)-1);

temp[sizeof(temp)-1] = '\0';

char \*token = strtok(temp, ",");

if (token) {

strncpy(firstName, token, sizeof(firstName)-1);

firstName[sizeof(firstName)-1] = '\0';

token = strtok(NULL, ",");

}

if (token) {

strncpy(lastName, token, sizeof(lastName)-1);

lastName[sizeof(lastName)-1] = '\0';

token = strtok(NULL, ",");

}

if (token) {

strncpy(ageStr, token, sizeof(ageStr)-1);

ageStr[sizeof(ageStr)-1] = '\0';

token = strtok(NULL, "");

}

if (token) {

strncpy(address, token, sizeof(address)-1);

address[sizeof(address)-1] = '\0';

}

printf("\n-----------------------------\n");

printf("Client %d\n", clientCount);

printf("Name : %s %s\n", firstName, lastName);

printf("Age : %s\n", ageStr);

printf("Address : %s\n", address);

printf("-----------------------------\n\n");

}

}

close(fd);

}

writeToLog(logFile, "Server stopped");

fclose(logFile);

}

/\*

\* FUNCTION: writeToLog

\* PROGRAMMER: Cy Iver Torrefranca & Tuan Thanh Nguyen

\* DESCRIPTION: Writes a message to the log file with a timestamp.

\* PARAMETERS:

\* FILE \*logFile : Pointer to the opened log file.

\* const char \*message : Message to log.

\* RETURNS : n/a

\*/

void writeToLog(FILE \*logFile, const char \*message) {

if (logFile) {

time\_t now;

time(&now);

char \*timeStr = ctime(&now);

// Remove newline from time string

if (timeStr) {

timeStr[strlen(timeStr) - 1] = '\0';

}

fprintf(logFile, "[%s] %s\n", timeStr ? timeStr : "Unknown time", message);

fflush(logFile);

}

}

/\*

\* FUNCTION: timeout\_handler

\* PROGRAMMER: Cy Iver Torrefranca

\* DESCRIPTION: Signal handler for SIGALRM. Terminates the server after timeout.

\* PARAMETERS:

\* int sig - Signal number (SIGALRM)

\* RETURNS: n/a (exits program)

\*/

void timeout\_handler(int sig) {

(void)sig; // Suppress unused parameter warning

printf("\nServer timeout: No activity for 2 minutes. Terminating server...\n");

// Log the timeout event

FILE \*logFile = fopen("travel\_agency.log", "a");

if (logFile) {

writeToLog(logFile, "Server terminated due to inactivity timeout (2 minutes)");

fclose(logFile);

}

exit(TIMEOUT\_SUCCESSFUL);

}

/\*

\* FUNCTION: reset\_timeout

\* PROGRAMMER: Cy Iver Torrefranca

\* DESCRIPTION: Resets the alarm timer to 2 minutes from current time.

\* PARAMETERS: n/a

\* RETURNS: n/a

\*/

void reset\_timeout(void) {

alarm(CANCEL\_TIMEOUT); // Cancel current alarm

alarm(TIMEOUT\_DURATION); // Reset to 2 minutes (120 seconds)

}