```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/*
    System's Programming Phase 2 Algorithm

    Author: Jesus M. Morales
    Due Date: 3/25/2018


*/

typedef struct
{
    char label[10];
    int memoryAddress;
}LABELS;

typedef struct
{
    char *label;
    char *mnemonic;
    char *opcode;
} TOKEN;

typedef struct
{
    char mnemonic[5];
    int opcode;
} OPCODE;

void breakupLine(char *input, char *command, char *param1, char *param2, int 
  *numParams);
void loadFile(char *fileName);
void passOne(char * fileName);
void passTwo();
void executeFile();
void debugFile();
void dumpFile();
void helpFile();
void assembleFile();
void errorFile();
int programLenght;

int main(void)
{
    char input[50];
    char command[50];
    char param1[50];
    char param2[50];

    printf("Hello welcome to Jesus Morales Personal Assembler\n \n");
    while (1)
    {
        int numParams = 0;
        int len = 0;
        printf("Command ----> ");
        fgets(input, 50, stdin);

        len = strlen(input) - 1;
        if (input[len] == '\n')
        {
            input[len] = '\0';
        }

        breakupLine(input, command, param1, param2, &numParams);
        numParams--;

        if (strcmp(command, "load") == 0)
        {
            if (numParams == 1)
            {
                loadFile(param1);
            }
            else
                errorFile();
        }
        else if (strcmp(command, "execute") == 0)
        {
            executeFile();
        }
        else if (strcmp(command, "debug") == 0)
        {
            debugFile();
        }
        else if (strcmp(command, "dump") == 0)
        {
            if (numParams == 2)
            {
                dumpFile();
            }
            else
                errorFile();
        }
        else if (strcmp(command, "help") == 0)
        {
            helpFile();
        }
        else if (strcmp(command, "assemble") == 0)
        {
            if (numParams == 1)
            {
                assembleFile();
            }
            else
                errorFile();
        }
        else if (strcmp(command, "dir") == 0)
        {
            system("dir");
        }
        else if (strcmp(command, "exit") == 0)
        {
            break;
        }
        else
        {
            printf("Invalid Command , for any help type 'help' to display the command list. \n \n");
        }

        numParams = 0;
    }
    return 0;
}

void breakupLine(char *input, char *command, char *param1, char *param2, int *numParams)
{
    command[0] = param1[0] = param2[0] = '\0';
    *numParams = sscanf(input, "%s %s %s %*s", command, param1, param2);
}

void loadFile(char *param1)
{
    printf("Loading file:  %s\n", param1);
    passOne(param1);
    passTwo();
    printf("The Programg lenght of this file is:   %d Bytes\n\n", programLenght);
    programLenght = 0;

}
void executeFile()
{
    printf(" is not yet avaibalbe.\n");
```

```c
}
void debugFile()
{
    printf("debug is not avaialabe.\n");
}
void dumpFile()
{
    printf("dump is not avaiblable.\n");
}
void helpFile()
{
    printf("\n");
    printf("\tWelcome to the Help menu. \n");
    printf("\tCommands are the following: \n \n");
    printf("\tload [file_name]\n");
    printf("\texecute \n");
    printf("\tdebug \n");
    printf("\tdump [start] [end] \n");
    printf("\thelp \n");
    printf("\tassemble [file_name] \n");
    printf("\tdirectory \n");
    printf("\texit \n\n");
    printf("\t**ALL COMMANDS ARE CASE SENSITIVE.**\n\n");
}
void assembleFile()
{
    printf("assemble not avaibalbe. \n");
}
void errorFile()
{
    printf("You typed the wrong number of parameters try again. \n");
}
void passOne(char *param1)
{
    char input[500];
    char *tokenizer = input;

    int start = 0;
    int locctr = 0;
    int memLenght = 0;
    int numLabels = 0;
    int numMnemonics = 25;
    int index = 0;

    int labelPresentFlag = 0;
    int duplicateLabelFlag = 0;
    int illegalLabelFlag = 0;
    int illegalOperationFlag = 0;
    int missingDataDirectiveFlag = 0;
    int missingStartFlag = 0;
    int missingEndFlag = 0;
    int tooManyLabelsFlag = 0;
    int programTooLongFlag = 0;
    int errorCode = 0;

    FILE *source_file, *symbol_file, *intermediate_file, *opcode_file;
    LABELS labelStructure[500];
    TOKEN tokenStructure;
    OPCODE opcodeStructure[] = { { "ADD", 0x18 },{ "AND",0x58 },{ "COMP", 0x28 }, { "DIV", 0x24 },
                                { "J", 0x3C },{ "JEQ", 0x30 },{ "JGT", 0x34 }, { "JLT", 0x38 },
                                { "JSUB", 0x48 },{ "LDA", 0x00 },{ "LDCH", 0x50 },{ "LDL", 0x08 },
                                { "LDX", 0x04 },{ "MUL", 0x20 },{ "OR", 0x44 }, { "RD", 0xD8 },
                                { "RSUB", 0x4C },{ "STA", 0x0C },{ "STCH", 0x54 },{ "STL", 0x14 },
                                { "STX", 0x10 },{ "SUB", 0x1C },{ "TD", 0xE0 }, { "TIX", 0x2C },{ "WD", 0xDC } };

    source_file = fopen(param1, "r");
    intermediate_file = fopen("intermediate.txt", "w");
    symbol_file = fopen("symbolTable.txt", "w");

    if (source_file == NULL)
    {
        printf("Error openning file does not exist: %s\n", param1);
        return;
    }

    tokenStructure.label = (char *)malloc(6);
    tokenStructure.mnemonic = (char *)malloc(6);
    tokenStructure.opcode = (char *)malloc(6);

    while (fgets(input, 500, source_file))
    {
        labelPresentFlag = 0;
        duplicateLabelFlag = 0;
        illegalLabelFlag = 0;
        illegalOperationFlag = 0;
        missingDataDirectiveFlag = 0;
        missingStartFlag = 0;
        missingEndFlag = 0;
        tooManyLabelsFlag = 0;
        programTooLongFlag = 0;
        errorCode = 0;
        memLenght = 0;

        /*  Check if label is present in the string line    */
        if (input[0] == ' ' || input[0] == '\t')
        {
            labelPresentFlag = 0;
        }
        else
        {
            labelPresentFlag = 1;
        }

        /*  Check if comment is present in the string line  */
        if (input[0] == '.')
        {
            continue;
        }

        /*  Tokenize the input string   */
        tokenizer = strtok(input, " \t\r\n\v\f");

        /*  Remove of the trailing newLine at the end of the string */
        int counter = 0;
        while (input[counter - 1] != '\n')
        {
            counter++;
        }
        input[counter] = '\0';

        /*  If there is a label */
        if (labelPresentFlag == 1)
        {
            /*  Tokenize the label into the structure   */
            strcpy(tokenStructure.label, tokenizer);

            /*  Tokenize the mnemonic into the structure    */
            tokenizer = strtok(NULL, " \t\r\n\v\f");
            strcpy(tokenStructure.mnemonic, tokenizer);

            /*  Tokenize the opcode of the mnemonic into the structure  */
            tokenizer = strtok(NULL, " \t\r\n\v\f");
            strcpy(tokenStructure.opcode, tokenizer);

            /*  Add the labels to the structure to create a list of existing
              labels/symbols */
            strcpy(labelStructure[numLabels].label, tokenStructure.label);
            labelStructure[numLabels].memoryAddress = locctr;

            /*  Check if there are labels in the list   */
            if (numLabels > 0)
```

```c
                {
                    /*  Check if limit of labels has been reached   */
                    if (numLabels > 500)
                    {
                        tooManyLabelsFlag = 1;
                    }

                    /*  Inefficiently scan the label/symbol list to check for duplicate labels/symbols     */
                    for (int i = 0; i < numLabels; i++)
                    {
                        if (strcmp(labelStructure[i].label, tokenStructure.label) == 0)
                        {
                            duplicateLabelFlag = 1;
                        }
                    }
                }

                /*  Check if the label is legal */
                if (!isalpha(tokenStructure.label[0]))
                {
                    illegalLabelFlag = 1;
                }

                /*  Check if we have a START directive in the beginning of the program */
                if (index == 0 && strcmp(tokenStructure.mnemonic, "START") != 0)
                {
                    missingStartFlag = 1;
                    locctr = 0;
                }

                /*  Check if we have a END directive in the end of the program  */
                if (missingEndFlag == 1 && errorCode == 0)
                {
                    if (strcmp(tokenStructure.mnemonic, "END") != 0)
                    {
                        missingEndFlag = 1;
                    }
                }

                /*  If directive START initialize LOCCTR to the starting address */
                if (strcmp(tokenStructure.mnemonic, "START") == 0) // if start directive initialize locct to the start(convert the string to integer)
                {
                    start = atoi(tokenStructure.opcode);
                    locctr = start;
                }

                /*  Check if program is too long    */
                if (locctr > 6700)
                {
                    programTooLongFlag = 1;
                }
                /* Lenght size in memory from the directives to increment the LOCCTR */
                if (strcmp(tokenStructure.mnemonic, "WORD") == 0)
                {
                    memLenght += 3;
                }
                if (strcmp(tokenStructure.mnemonic, "RESB") == 0)
                {
                    memLenght += atoi(tokenStructure.opcode);

                }
                if (strcmp(tokenStructure.mnemonic, "RESW") == 0)
                {
                    memLenght += 3 * atoi(tokenStructure.opcode);
                }

                if (strcmp(tokenStructure.mnemonic, "BYTE") == 0)
                {
                    /*  Check if operand is set to read a string (C) or a hexadecimal (X)    */
                    if (tokenStructure.opcode[0] == 'C')
                    {

                        int bufferSpace = 0;
                        int counter = 2;
                        while (tokenStructure.opcode[counter] != '\'' && bufferSpace < 30)
                        {
                            bufferSpace++;
                            counter++;
                        }
                        memLenght += bufferSpace;

                    }
                    else if (tokenStructure.opcode[0] == 'X')
                    {
                        char hexInput[16];
                        int bufferSpace = 0;
                        int counter = 2;
                        while (tokenStructure.opcode[counter] != '\'' && bufferSpace < 16)
                        {
                            hexInput[bufferSpace] = tokenStructure.opcode[counter];
                            bufferSpace++;
                            counter++;
                        }
                        hexInput[bufferSpace] = '\0';
                        memLenght = (int)strtol(hexInput, NULL, 16);
                    }

                    /* Check for errors in the input for the BYTE directive */
                    else
                    {
                        illegalOperationFlag = 1;
                    }

                    if (tokenStructure.opcode[1] != '\'' || tokenStructure.opcode[strlen(tokenStructure.opcode) - 1] != '\'')
                    {
                        missingDataDirectiveFlag = 1;
                    }
                }

                /* Error Flag conditions    */
                if (duplicateLabelFlag == 1 && errorCode == 0)
                {
                    errorCode = 1;
                }
                else if (illegalLabelFlag == 1 && errorCode == 0)
                {
                    errorCode = 2;
                }
                else if (illegalOperationFlag == 1 && errorCode == 0)
                {
                    errorCode = 3;
                }
                else if (missingDataDirectiveFlag == 1 && errorCode == 0)
                {
                    errorCode = 4;
                }
                else if (missingStartFlag == 1 && errorCode == 0)
                {
                    errorCode = 5;
                }
                else if (missingEndFlag == 1 && errorCode == 0)
                {
                    errorCode = 6;
                }
                else if (tooManyLabelsFlag == 1 && errorCode == 0)
                {
                    errorCode = 7;
                }
                else if (programTooLongFlag == 1 && errorCode == 0)
                {
                    errorCode = 8;
                }

                /*Print to the intermediate file and symbol file */
                fprintf(intermediate_file, "%d\t%s\t%s\t%s\t%d\n", locctr, 
                  tokenStructure.label, tokenStructure.mnemonic, 
                  tokenStructure.opcode, errorCode);
                fprintf(symbol_file, "%d\t %s\n", locctr, tokenStructure.label);

                /* Search for the mnemonic in the opcode table and add 3 to it  */
                for (int i = 0; i < numMnemonics; i++)
                {
                    if (strcmp(opcodeStructure[i].mnemonic, tokenStructure.mnemonic) 
                      == 0)
                    {
                        locctr += 3;
                    }
                }

                /*Update the memory locations after LOCCTR is printed in the file 
                  */
                if (strcmp(tokenStructure.mnemonic, "BYTE") == 0 || strcmp
                  (tokenStructure.mnemonic, "RESB") == 0 || strcmp
                  (tokenStructure.mnemonic, "RESW") == 0 || strcmp
                  (tokenStructure.mnemonic, "WORD") == 0)
                {
                    locctr += memLenght;
                }

                /*  Increment the number of labels in the system    */
                numLabels++;
            }

            /*If there is no label in the input line do the same as above but without 
               labels */
            else
            {
                /*  Tokenize the mnemonic into the structure    */
                strcpy(tokenStructure.mnemonic, tokenizer);

                /*  Tokenize the opcode into the structure  */
                tokenizer = strtok(NULL, " \t\r\n\v\f");
                strcpy(tokenStructure.opcode, tokenizer);

                /*  Check if we have a START directive in the beginning of the 
```

```c
program */
        if (index == 0 && strcmp(tokenStructure.mnemonic, "START") != 0)
        {
            missingStartFlag = 1;
            locctr = 0;
        }

        /*  Check if we have a END directive in the end of the program  */
        if (missingEndFlag == 1 && errorCode == 0)
        {
            if (strcmp(tokenStructure.mnemonic, "END") != 0)
            {
                missingEndFlag = 1;
            }
        }

        /*  If directive START initialize LOCCTR to the starting address      */
        if (strcmp(tokenStructure.mnemonic, "START") == 0) // if start directive initialize locct to the start(convert the string to integer)
        {
            start = atoi(tokenStructure.opcode);
            locctr = start;
        }

        /*  Check if program is too long    */
        if (locctr > 32000)
        {
            programTooLongFlag = 1;
        }

        /* Lenght size in memory from the directives to increment the LOCCTR  */
        if (strcmp(tokenStructure.mnemonic, "WORD") == 0)
        {
            memLenght += 3;
        }
        if (strcmp(tokenStructure.mnemonic, "RESB") == 0)
        {
            memLenght += atoi(tokenStructure.opcode);

        }
        if (strcmp(tokenStructure.mnemonic, "RESW") == 0)
        {
            memLenght += 3 * atoi(tokenStructure.opcode);
        }

        if (strcmp(tokenStructure.mnemonic, "BYTE") == 0)
```

```
                {
                    /*  Check if operand is set to read a string (C) or a hexadecimal (X)    */
                    if (tokenStructure.opcode[0] == 'C')
                    {

                        int bufferSpace = 0;
                        int counter = 2;
                        while (tokenStructure.opcode[counter] != '\'' && bufferSpace < 30)
                        {
                            bufferSpace++;
                            counter++;
                        }
                        memLenght += bufferSpace;

                    }
                    else if (tokenStructure.opcode[0] == 'X')
                    {
                        char hexInput[16];
                        int bufferSpace = 0;
                        int counter = 2;
                        while (tokenStructure.opcode[counter] != '\'' && bufferSpace < 16)
                        {
                            hexInput[bufferSpace] = tokenStructure.opcode[counter];
                            bufferSpace++;
                            counter++;
                        }
                        hexInput[bufferSpace] = '\0';
                        memLenght = (int)strtol(hexInput, NULL, 16);
                    }

                    /* Check for errors in the input for the BYTE directive */
                    else
                    {
                        illegalOperationFlag = 1;
                    }

                    if (tokenStructure.opcode[1] != '\'' || tokenStructure.opcode[strlen(tokenStructure.opcode) - 1] != '\'')
                    {
                        missingDataDirectiveFlag = 1;
                    }
                }

                /* Error Flag conditions    */
                if (duplicateLabelFlag == 1 && errorCode == 0)
                {
```

```c
                errorCode = 1;
            }
            else if (illegalLabelFlag == 1 && errorCode == 0)
            {
                errorCode = 2;
            }
            else if (illegalOperationFlag == 1 && errorCode == 0)
            {
                errorCode = 3;
            }
            else if (missingDataDirectiveFlag == 1 && errorCode == 0)
            {
                errorCode = 4;
            }
            else if (missingStartFlag == 1 && errorCode == 0)
            {
                errorCode = 5;
            }
            else if (missingEndFlag == 1 && errorCode == 0)
            {
                errorCode = 6;
            }
            else if (tooManyLabelsFlag == 1 && errorCode == 0)
            {
                errorCode = 7;
            }
            else if (programTooLongFlag == 1 && errorCode == 0)
            {
                errorCode = 8;
            }

            /*Print to the intermediate file and symbol file */
            fprintf(intermediate_file, "%d\t\t\t%s\t%s\t%d\n", locctr, tokenStructure.mnemonic, tokenStructure.opcode, errorCode);

            /* Search for the mnemonic in the opcode table and add 3 to it  */
            for (int i = 0; i < numMnemonics; i++)
            {
                if (strcmp(opcodeStructure[i].mnemonic, tokenStructure.mnemonic) == 0)
                {
                    locctr += 3;
                }
            }

            /*Update the memory locations after LOCCTR is printed in the file */
            if (strcmp(tokenStructure.mnemonic, "BYTE") == 0 || strcmp(tokenStructure.mnemonic, "RESB") == 0 || strcmp
              (tokenStructure.mnemonic, "RESW") == 0 || strcmp
              (tokenStructure.mnemonic, "WORD") == 0)
            {
                locctr += memLenght;
            }
        }
        index++;
    }

    programLenght = locctr - start;
    printf("Pass One complete successfully. \n");

    fprintf(intermediate_file, "\n\n\t Printing Error Code List: \n\n");
    fprintf(intermediate_file,
      "*=================================================*\n");
    fprintf(intermediate_file, "\tNo Error = 0\n");
    fprintf(intermediate_file, "\tDuplicate Label = 1\n");
    fprintf(intermediate_file, "\tIllegal Label = 2\n");
    fprintf(intermediate_file, "\tIllegal Operation = 3\n");
    fprintf(intermediate_file, "\tIllegal Data Storage Directive = 4\n");
    fprintf(intermediate_file, "\tMissing START Directive = 5\n");
    fprintf(intermediate_file, "\tMissing END Directive = 6\n");
    fprintf(intermediate_file, "\tToo Many Symbols = 7\n");
    fprintf(intermediate_file, "\tProgram Too Long = 8\n");
    fprintf(intermediate_file,
      "*=================================================*\n");

    fclose(intermediate_file);
    fclose(source_file);
    fclose(symbol_file);
}
void passTwo()
{
    printf("Pass Two is still in development. \n\n");
}

```