FORMAN CHRISTIAN COLLEGE
(A CHARTERED UNIVERSITY)

COMP 451 COMPILER CONSTRUCTION B

SPRING 2025

Assignment 1

C Preprocessor

Muhammad Zeeshan 261940660

Introduction:

This assignment focuses on simulating a preprocessor like code that traverses a c code file and transform that C code file into a new file where necessary adjustments are made. The adjustments to be made are:

Removing Blank Lines:

The sample input files contain unnecessary blank lines. The logic I have used to identify blank lines is identifying the placement of \n and ‘ ‘ and \t. If a line has this after we have created a newline then it is assumed that it is a blank line.

Pseudo Code explanation:

```
If newline
  If character after newline is \n or ‘ ‘ \t:
 Skip over it
  Else:
   Write the content in output file (these are not blank lines)
```

Removing Comments:

To remove comments the logic is simple.

We shall traverse the file using fgetc which gets a singular character.

We store this in a variable and check for the next character that indicates whether it’s a single or multi line comment.

Explanation can be found under the code.

Macro Expansion:

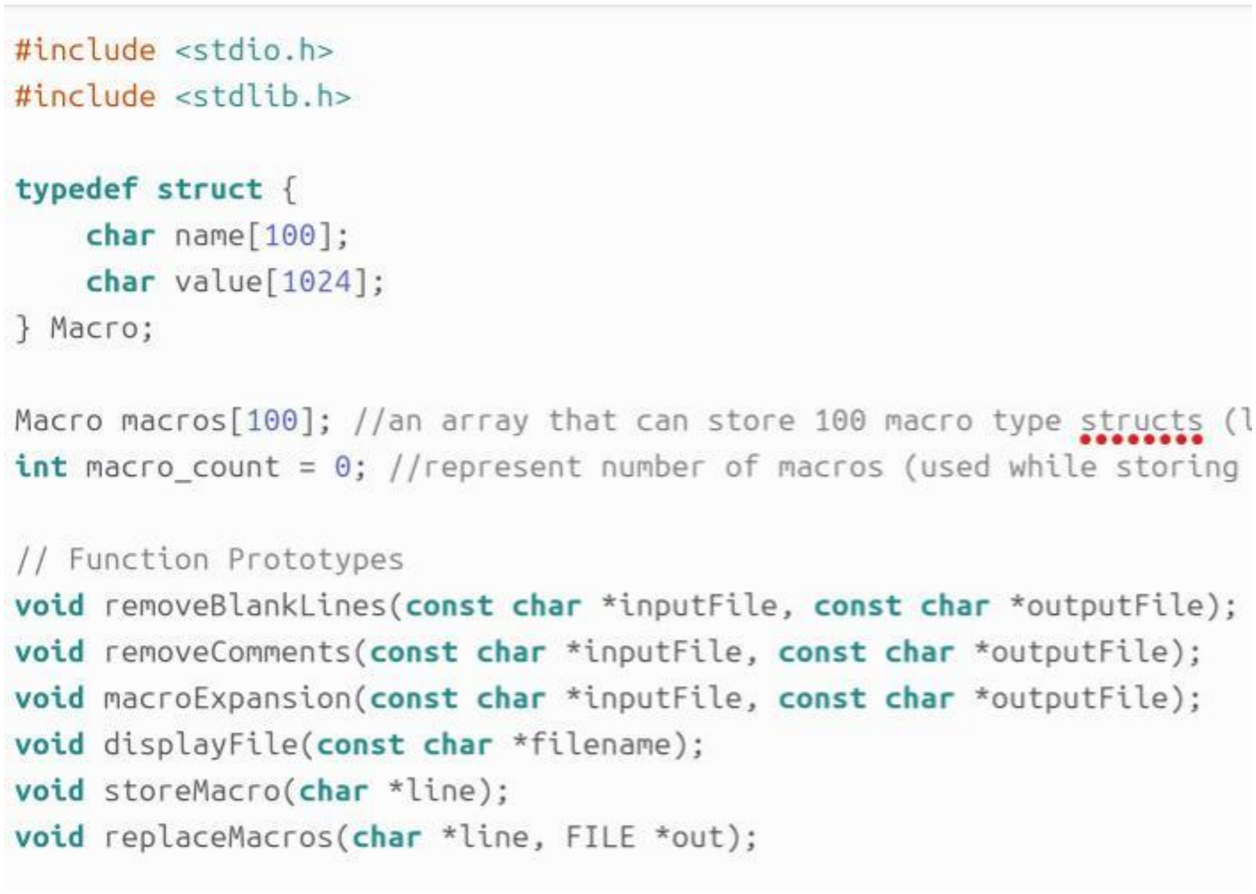
We will divide this task into two parts one where we store the macro and then replace the macro with its value/body

We will do this by defining a struct that has attributes name and value.

The name will be a macro and name that value corresponding to that struct will be the value define to the macro.

All these are the functionalities we are expected to cover in the code.

Code Explanation:

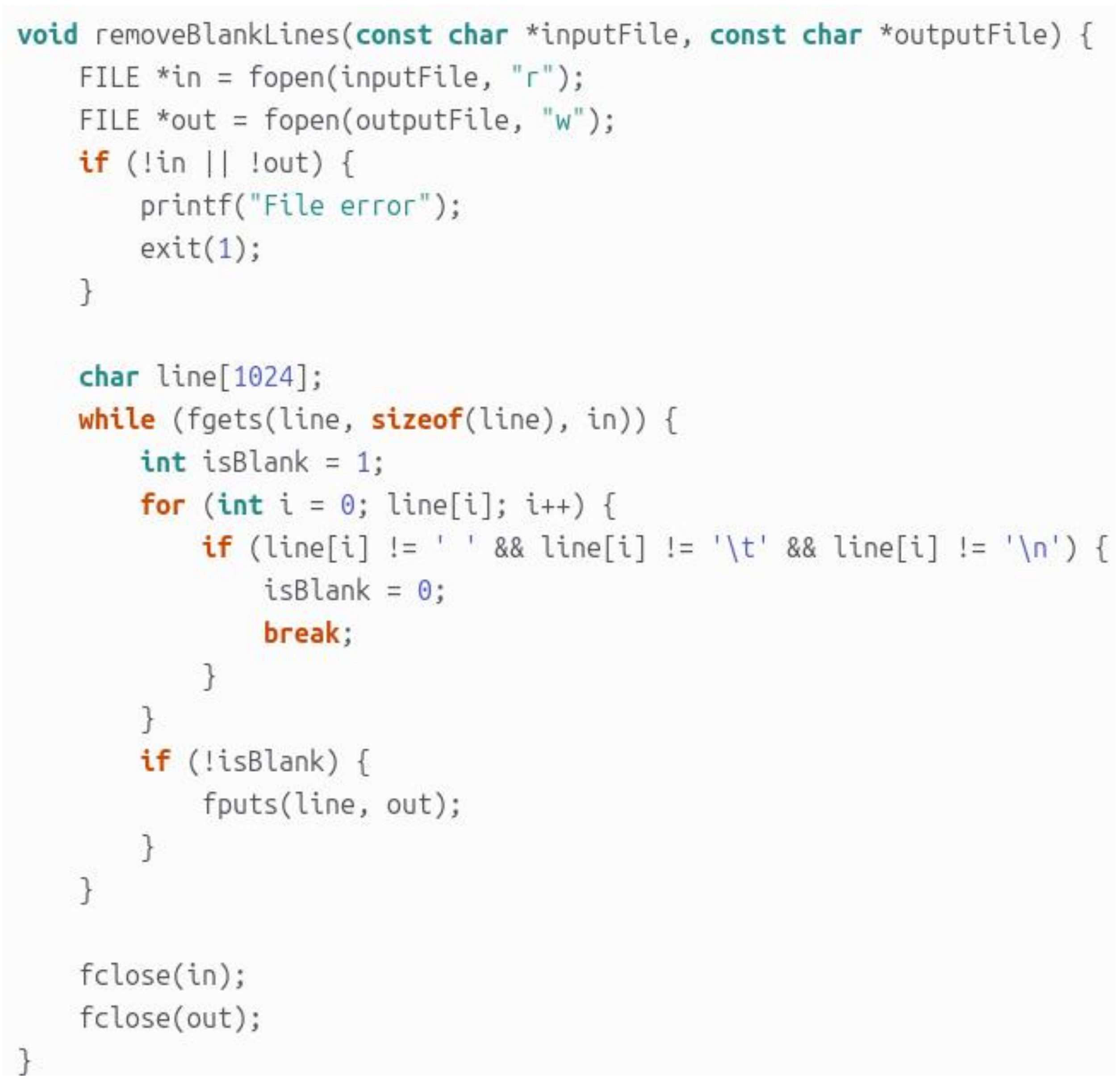


```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char name[100];
    char value[1024];
} Macro;

Macro macros[100]; //an array that can store 100 macro type structs (l
int macro_count = 0; //represent number of macros (used while storing

// Function Prototypes
void removeBlankLines(const char *inputFile, const char *outputFile);
void removeComments(const char *inputFile, const char *outputFile);
void macroExpansion(const char *inputFile, const char *outputFile);
void displayFile(const char *filename);
void storeMacro(char *line);
void replaceMacros(char *line, FILE *out);
```

- Including basic header files
- Defining a struct like explained in the logic of macro expansion.
- Creating an array of 100 macro struct type. This array can store 100 macro (we are limiting our code to handle 100 macros)
- Macro count increases as macros are stored. Useful when figuring out how many macros we have while traversing through them at the time when we want to replace macros
- Defining function prototype (displayFile,storeMacro and replaceMacris are helper functions for the main required functions)

Let us focus on each function first and then we will see how it is used in main.

Removing Blank Lines:



```c
void removeBlankLines(const char *inputFile, const char *outputFile) {
    FILE *in = fopen(inputFile, "r");
    FILE *out = fopen(outputFile, "w");
    if (!in || !out) {
        printf("File error");
        exit(1);
    }

    char line[1024];
    while (fgets(line, sizeof(line), in)) {
        int isBlank = 1;
        for (int i = 0; line[i]; i++) {
            if (line[i] != ' ' && line[i] != '\t' && line[i] != '\n') {
                isBlank = 0;
                break;
            }
        }
        if (!isBlank) {
            fputs(line, out);
        }
    }

    fclose(in);
    fclose(out);
}
```

- Accepts inputfile name and output filename.
- This output filename is not the final file name as the final file will go through other operations as well.
- So this file will be intermediary and is called noblank.c
- Opening inputfile to read and output file to write.

- Checking if both files were opened.
- Creating a buffer called line to store lines of the file.
- Traversing the file line by line using fgets , where line is buffer, size of line is the size, and the file traversed is in (input file)
- Initially setting a line to blank.
- Traversing through the character of the line using for loop.
- If the character is not just spaces or tab or newline, then isBlank flag is turned to false meaning line is not blank.

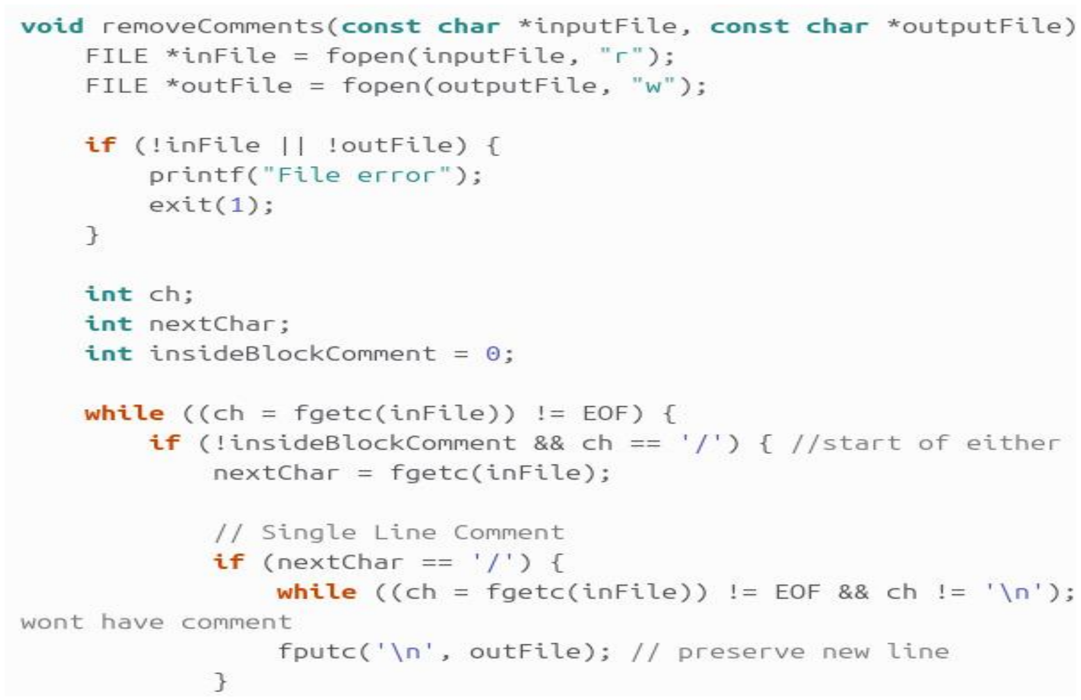
Why this works?

We are focusing on each line through fgets and we omit the very last character which is always \n. Since size of line automatically becomes size of single line of actual file it will always be one less than the number of characters plus \n.If \n occurred before that , that means the line has nothing else.

Our for loop is just detecting if a non blank character exists in the line if it does then it changes flag value.

- When the inner for loop detects a non blank line it breaks
- Control moves to the if block below just making sure the IsBlank flag is false stating the line in the buffer is safe/ not blank and just adding it to the output file.
- Close both input and output file.

Removing Comments:



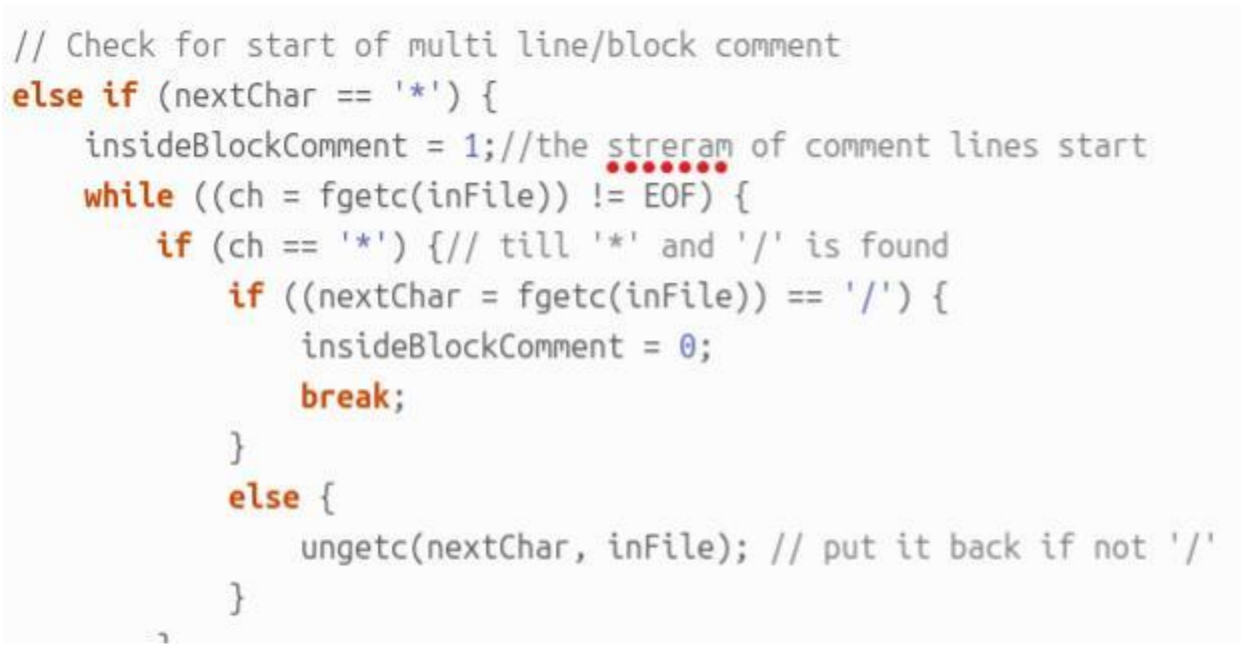
```
void removeComments(const char *inputFile, const char *outputFile)
    FILE *inFile = fopen(inputFile, "r");
    FILE *outFile = fopen(outputFile, "w");

    if (!inFile || !outFile) {
        printf("File error");
        exit(1);
    }

    int ch;
    int nextChar;
    int insideBlockComment = 0;

    while ((ch = fgetc(inFile)) != EOF) {
        if (!insideBlockComment && ch == '/') { //start of either
            nextChar = fgetc(inFile);

            // Single Line Comment
            if (nextChar == '/') {
                while ((ch = fgetc(inFile)) != EOF && ch != '\n');
wont have comment
                fputc('\n', outFile); // preserve new line
            }
```

- Same file opening logic
- Declaring variable of current character, next character and flag detecting if multiline comments are existing at the moment.
- Getting file contents character by character till EOF.
- Initially not assuming multiline comments and if current character is / it indicates a comment has started but which kind of comment depends on the next character.

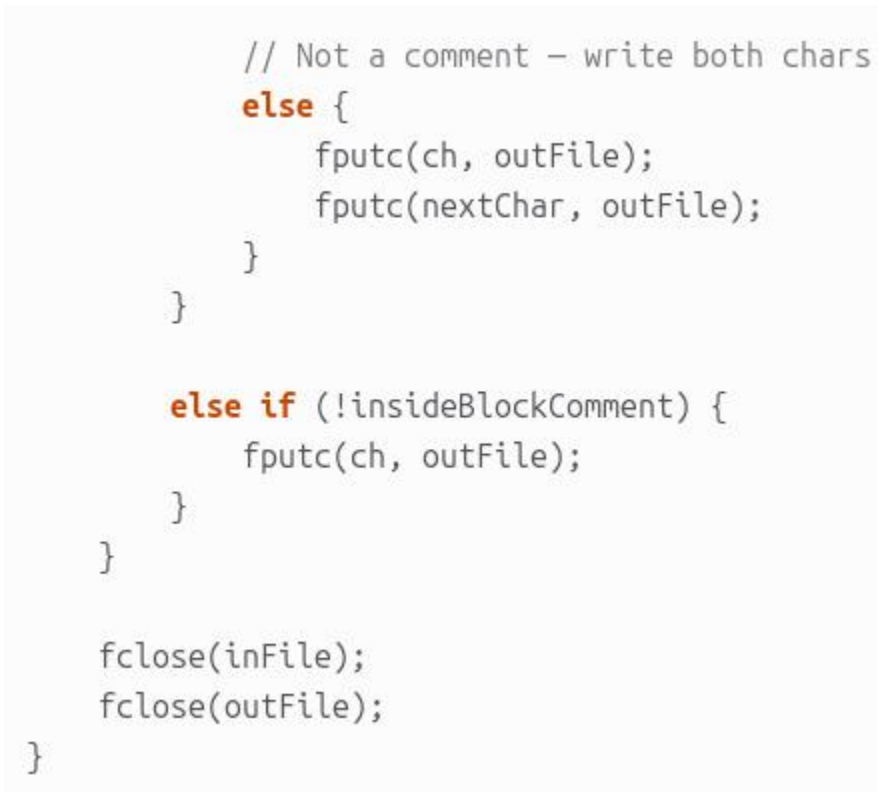
Why it works?

The nextchar will start reading file from where ch left off as reading files is sequential .

- If the nextchar is / it means single line comment.
- Moving ch forward to ignore the comment till EOF or \n is reached.
- Preserving newline character as it directly deals with how the code has been formatted.


```
// Check for start of multi line/block comment
else if (nextChar == '*') {
    insideBlockComment = 1;//the strseram of comment lines start
    while ((ch = fgetc(inFile)) != EOF) {
        if (ch == '*') {// till '*' and '/' is found
            if ((nextChar = fgetc(inFile)) == '/') {
                insideBlockComment = 0;
                break;
            }
            else {
                ungetc(nextChar, inFile); // put it back if not '/'
            }
```

- Else if next character is * stating multiline comments.
- Changing flag to 1
- Moving ch forward.
- If ch is * it could mean the multiline comment is about to end so check nextchar.
- If it is / that means multiline comments have been ignore so break the loop in order to allow outer while loop to handle further content in the file.
- Putting the accidentally stored character in nextchar back to input stream to allow ch to further track the content if the next character was not /.



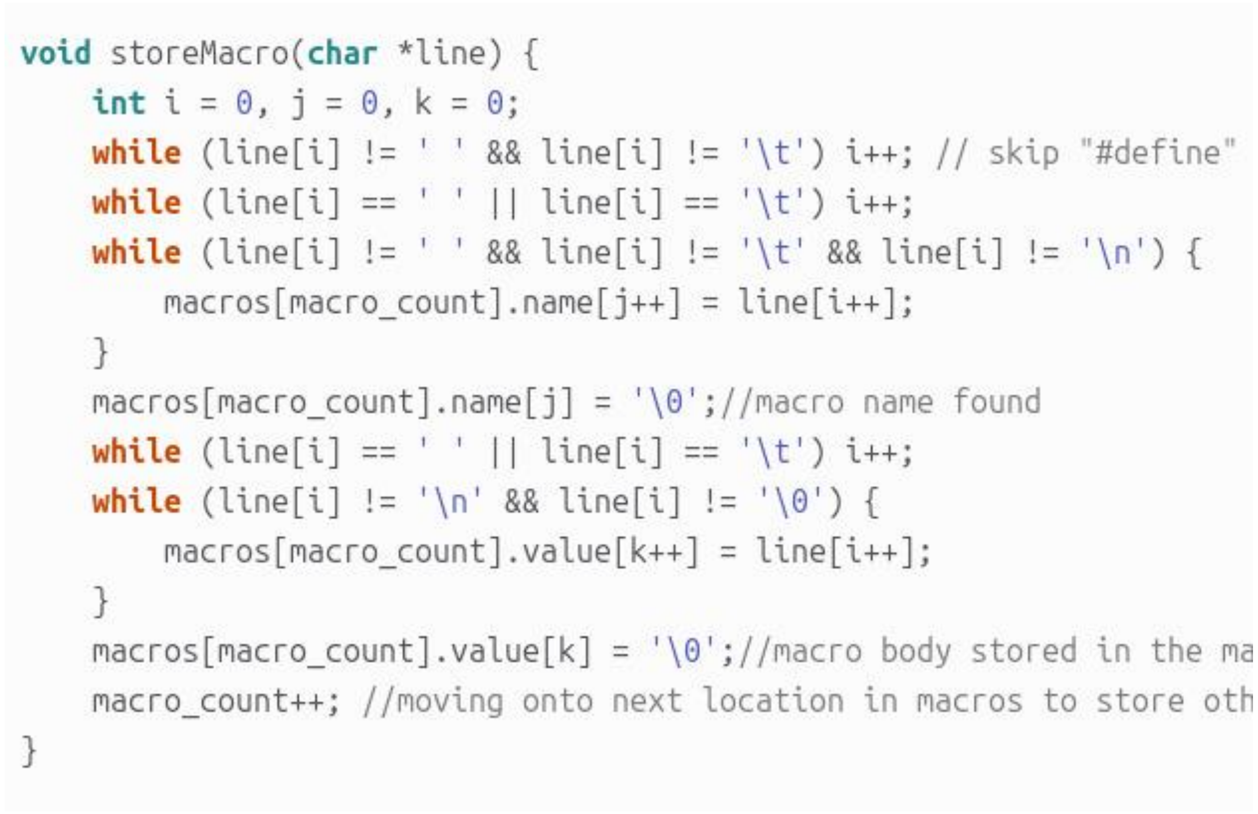
```
            // Not a comment - write both chars
            else {
                fputc(ch, outFile);
                fputc(nextChar, outFile);
            }
        }

        else if (!insideBlockComment) {
            fputc(ch, outFile);
        }
    }

    fclose(inFile);
    fclose(outFile);
}
```

- If ch was not / just put whatever was stored ch and nextchar sequentially in the output file.
- If the flag of multiline comment became 0 meaning the ch is not in a multiline comment, then put the current character in ch on output file.
- This output file is still intermediary and will behandled further by macro expansion.
- Close input and output files.

Expanding Macros Code:

Expanding Macros is comprised of 2 parts storing and replacing.

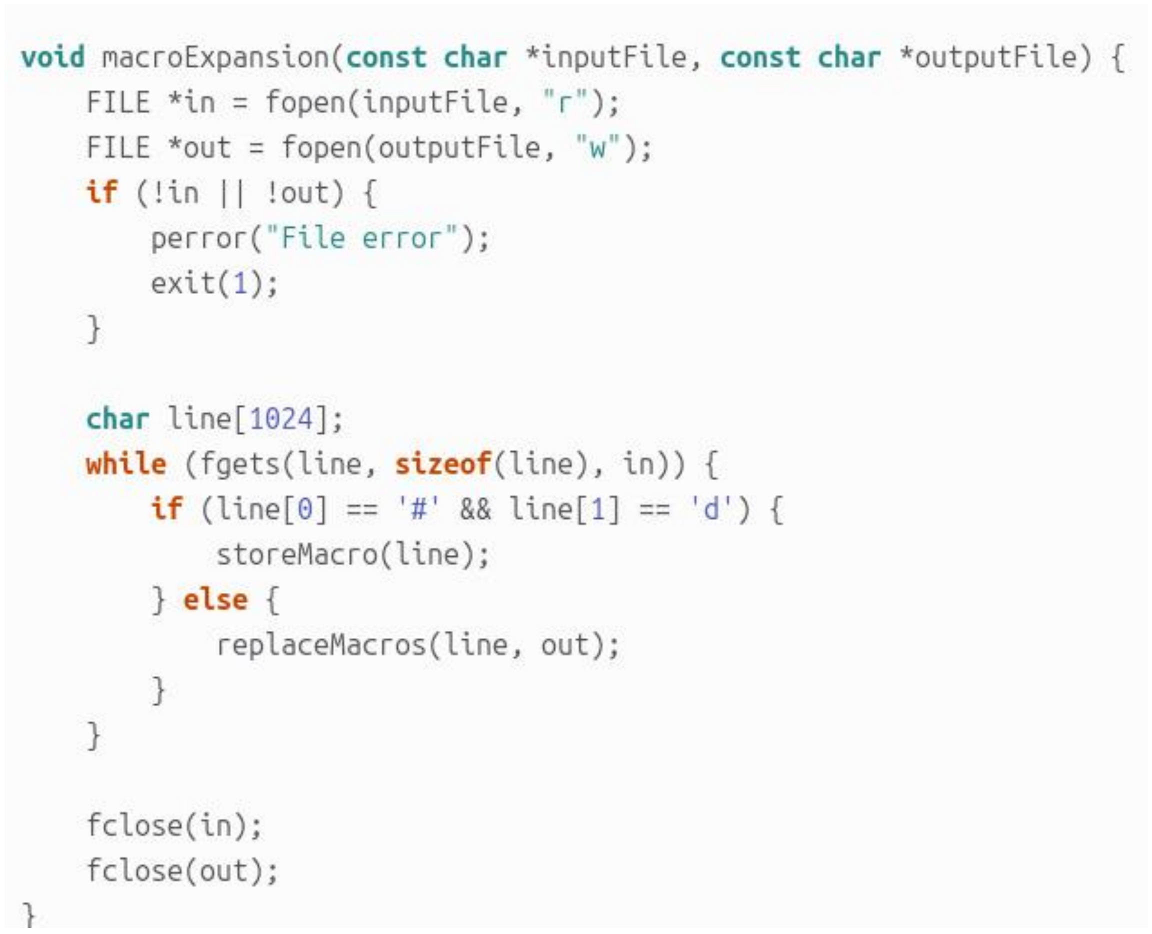
Storing Macro:



```c
void storeMacro(char *line) {
    int i = 0, j = 0, k = 0;
    while (line[i] != ' ' && line[i] != '\t') i++; // skip "#define"
    while (line[i] == ' ' || line[i] == '\t') i++;
    while (line[i] != ' ' && line[i] != '\t' && line[i] != '\n') {
        macros[macro_count].name[j++] = line[i++];
    }
    macros[macro_count].name[j] = '\0';//macro name found
    while (line[i] == ' ' || line[i] == '\t') i++;
    while (line[i] != '\n' && line[i] != '\0') {
        macros[macro_count].value[k++] = line[i++];
    }
    macros[macro_count].value[k] = '\0';//macro body stored in the ma
    macro_count++; //moving onto next location in macros to store oth
}
```

- The line passed to this function assumed that it containes #define.
- This part is done in the expandingmacros function.
- Skip define while skipping over the character of the line which are not blank spaces.
- Then skip over a blank space when it occurs.
- Then we are at the part of line that has a macro name.
- Store that macro name at the current space in macros array. The line till /n is the macro name and is stored character by character in char name.
- Making the last character of name \0 as it is a string
- Now move over the space.
- Now till line is not \n we have the body of the macro/ value.
- Store that value in the value attributes of the current struct we are processing in macros array.
- Make the last character of value \0
- Move to the next struct in the array and indicate how many structs there are.

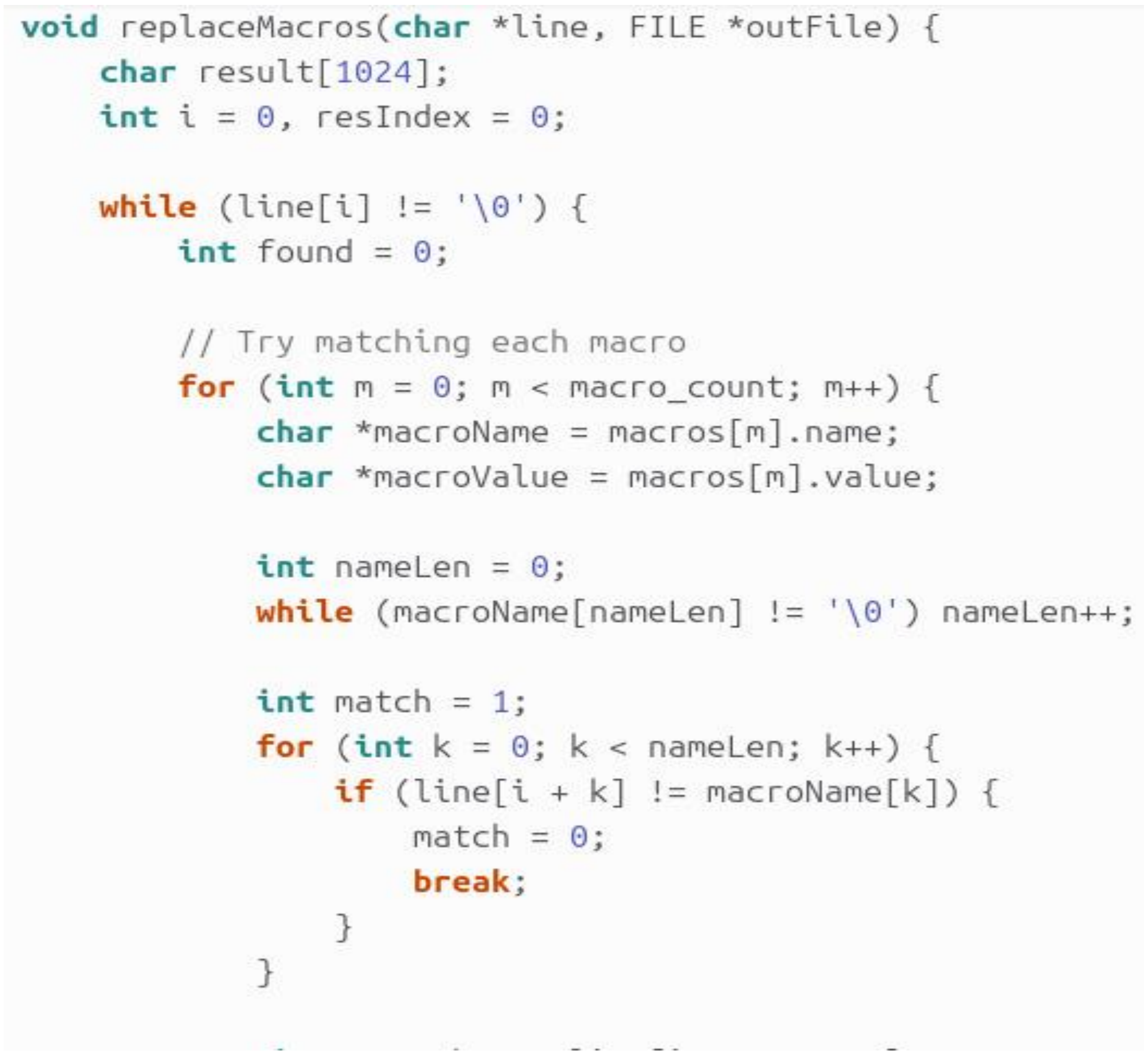
To make more sense of the logic where we replace macrod with their body let us take a look at our macroExpansion function that controls when these operations happens.



```c
void macroExpansion(const char *inputFile, const char *outputFile) {
    FILE *in = fopen(inputFile, "r");
    FILE *out = fopen(outputFile, "w");
    if (!in || !out) {
        perror("File error");
        exit(1);
    }

    char line[1024];
    while (fgets(line, sizeof(line), in)) {
        if (line[0] == '#' && line[1] == 'd') {
            storeMacro(line);
        } else {
            replaceMacros(line, out);
        }
    }

    fclose(in);
    fclose(out);
}
```

- Expects input and output file.
- Same logic for checking files
- Created buffer
- Traversing file using fgets
- If first character of line is # and second is d that means it is a macro defining statement so store it using storeMacro
- Else replace macro in the line and store appropriate and correct line on output file where macro is replaced by its actual value.
- Close both files.

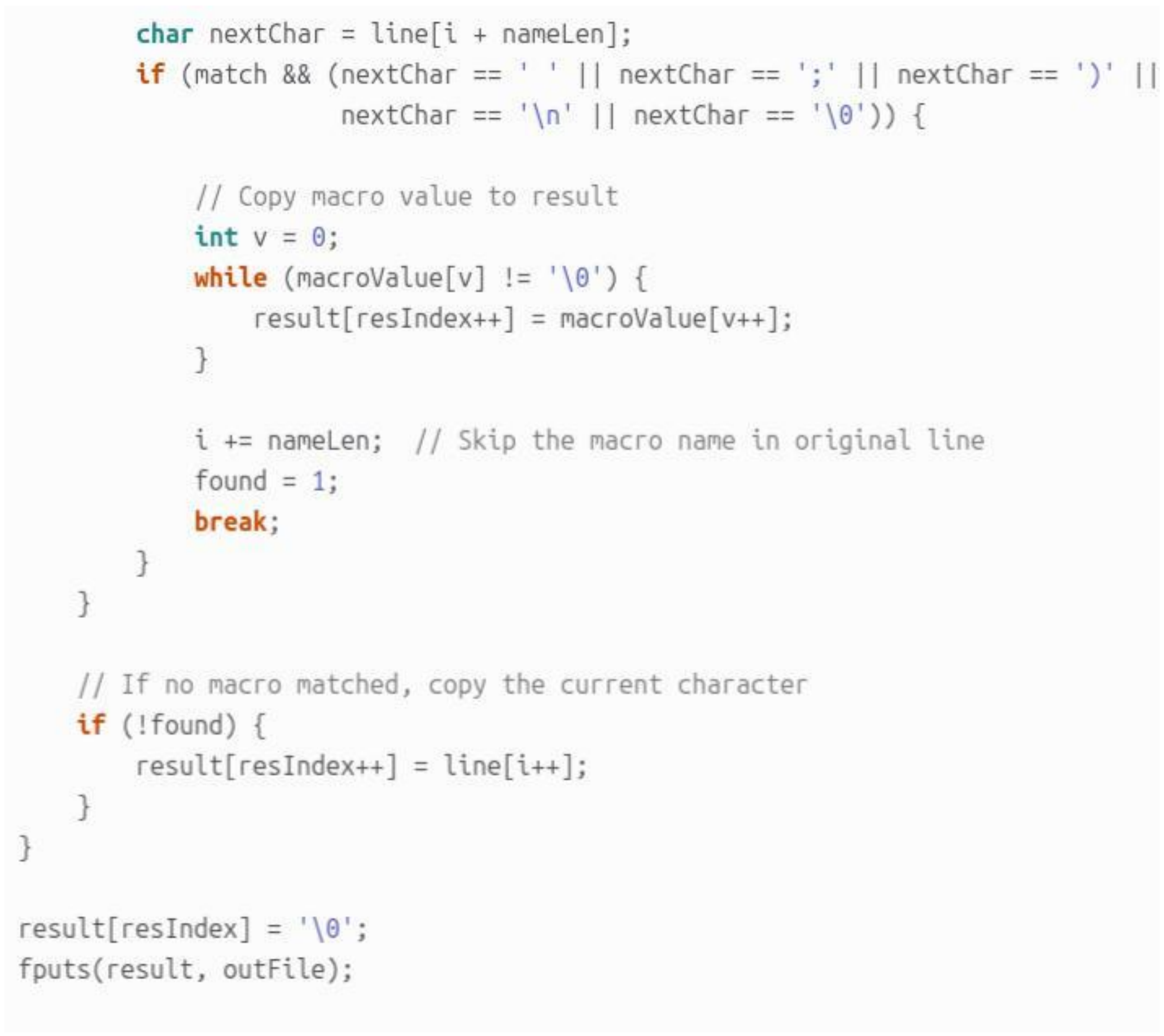
Replacing Macros:



```
void replaceMacros(char *line, FILE *outFile) {
    char result[1024];
    int i = 0, resIndex = 0;

    while (line[i] != '\0') {
        int found = 0;

        // Try matching each macro
        for (int m = 0; m < macro_count; m++) {
            char *macroName = macros[m].name;
            char *macroValue = macros[m].value;

            int nameLen = 0;
            while (macroName[nameLen] != '\0') nameLen++;

            int match = 1;
            for (int k = 0; k < nameLen; k++) {
                if (line[i + k] != macroName[k]) {
                    match = 0;
                    break;
                }
            }
```

- Accepts line and file to which write the result.
- Buffer for result
- Index counter for line and result index counter
- Traverse line character by character and found to 0.
- For loop going through array of macros to see if the macro matches with anything on our line. This can run many times if the macro in the line is further down the macros array
- Calculating length of macro name of all individual macros on every iteration.

- Assume it’s a match, using a for loop traverse the macroname character by character.
- Line is current still at its original position. Stretch out where line is at because we are checking if it is currently on a macro.
- Till k runs out and macro name is fully traversed and all characters matched that means it is a match so move on from for loop.
- If not then match is 0 and break the for loop as it was propbably not a macro and a necessary statement. The control moves to outer for loop to check matches again and again with other macros.



```
        char nextChar = line[i + nameLen];
        if (match && (nextChar == ' ' || nextChar == ';' || nextChar == ')' ||
                      nextChar == '\n' || nextChar == '\0')) {

            // Copy macro value to result
            int v = 0;
            while (macroValue[v] != '\0') {
                result[resIndex++] = macroValue[v++];
            }

            i += nameLen;  // Skip the macro name in original line
            found = 1;
            break;
        }
    }

    // If no macro matched, copy the current character
    if (!found) {
        result[resIndex++] = line[i++];
    }
}

result[resIndex] = '\0';
fputs(result, outFile);
```

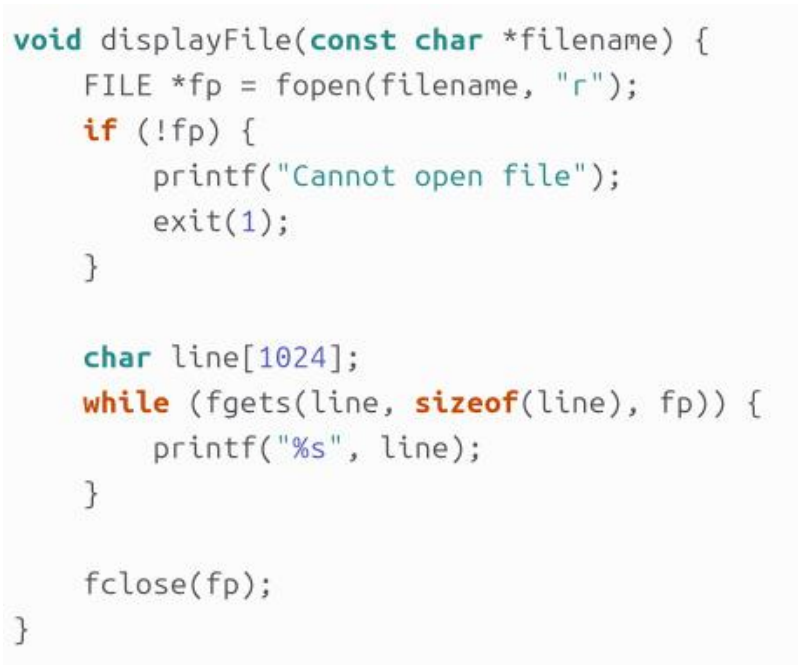
- Creating a variable nextChar , at the position we are after adding length of macro name.
- We have found that it is a match, and the value of nextChar is likely a space or parenthesis or the other described above.
- We have skipped over the macro and will now replace it with the value.
- Initializing v to traverse value attribute.

Why it works?

We are traversing and skipping the parts of the line we detected a s match with a macro by applying logic that was matching strings. While this is happening if found is 0 and there was no condition where a macro matched that likely suggests the macro is not here yet so we keep writing in the results and store the line in result in the output file using fputs.

- Changing flag of found to 1 suggesting macro has been found and then the whole process repeats again. The corrected value of macro is put in result not the macro which is the original line.

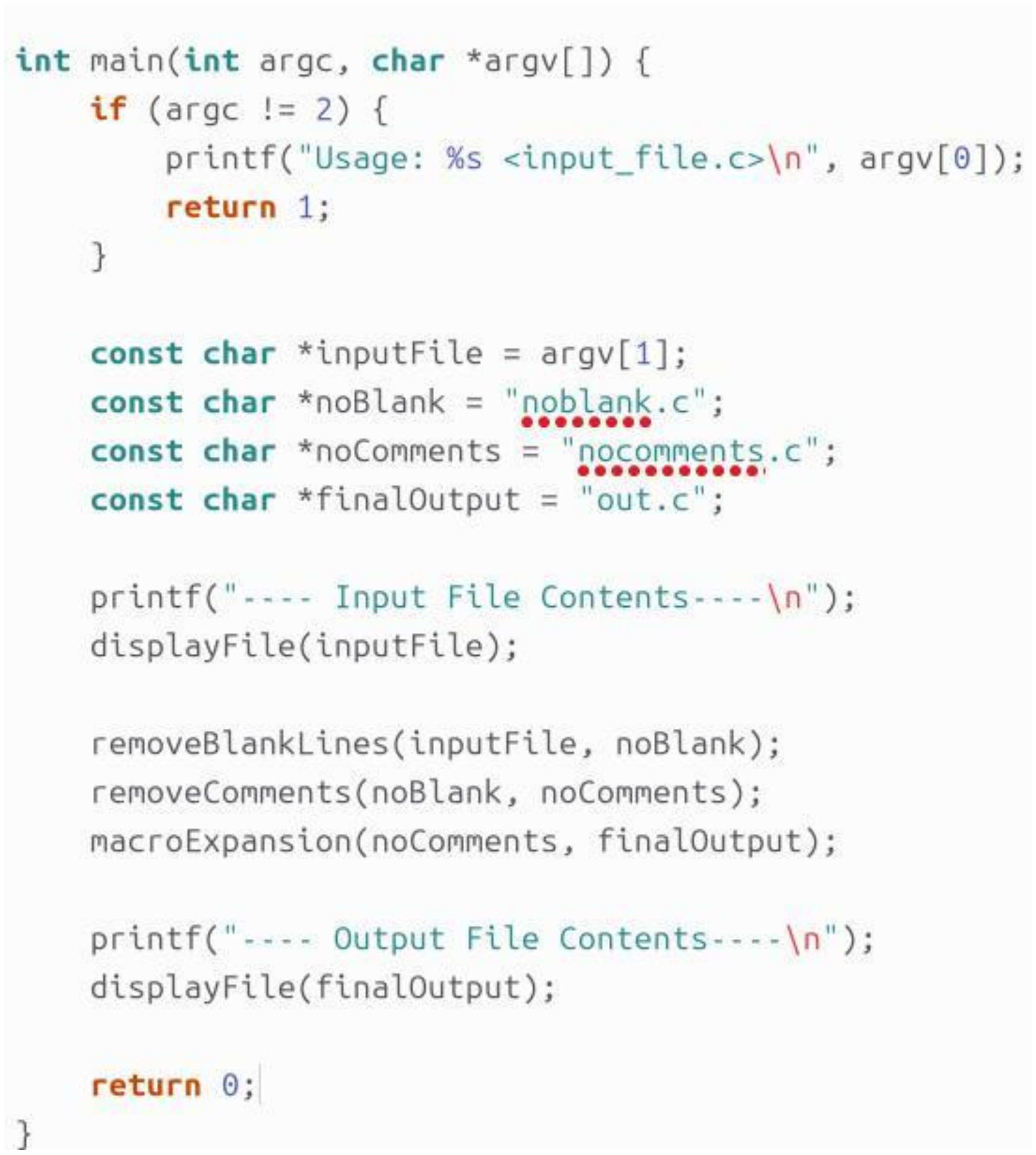
Display File:



```
void displayFile(const char *filename) {
    FILE *fp = fopen(filename, "r");
    if (!fp) {
        printf("Cannot open file");
        exit(1);
    }

    char line[1024];
    while (fgets(line, sizeof(line), fp)) {
        printf("%s", line);
    }

    fclose(fp);
}
```

- Simple function expect a file name to display
- Stores it in file pointer.
- Creates a buffer
- Traverse file line by line
- Prints file line by line
- Close file

MAIN():

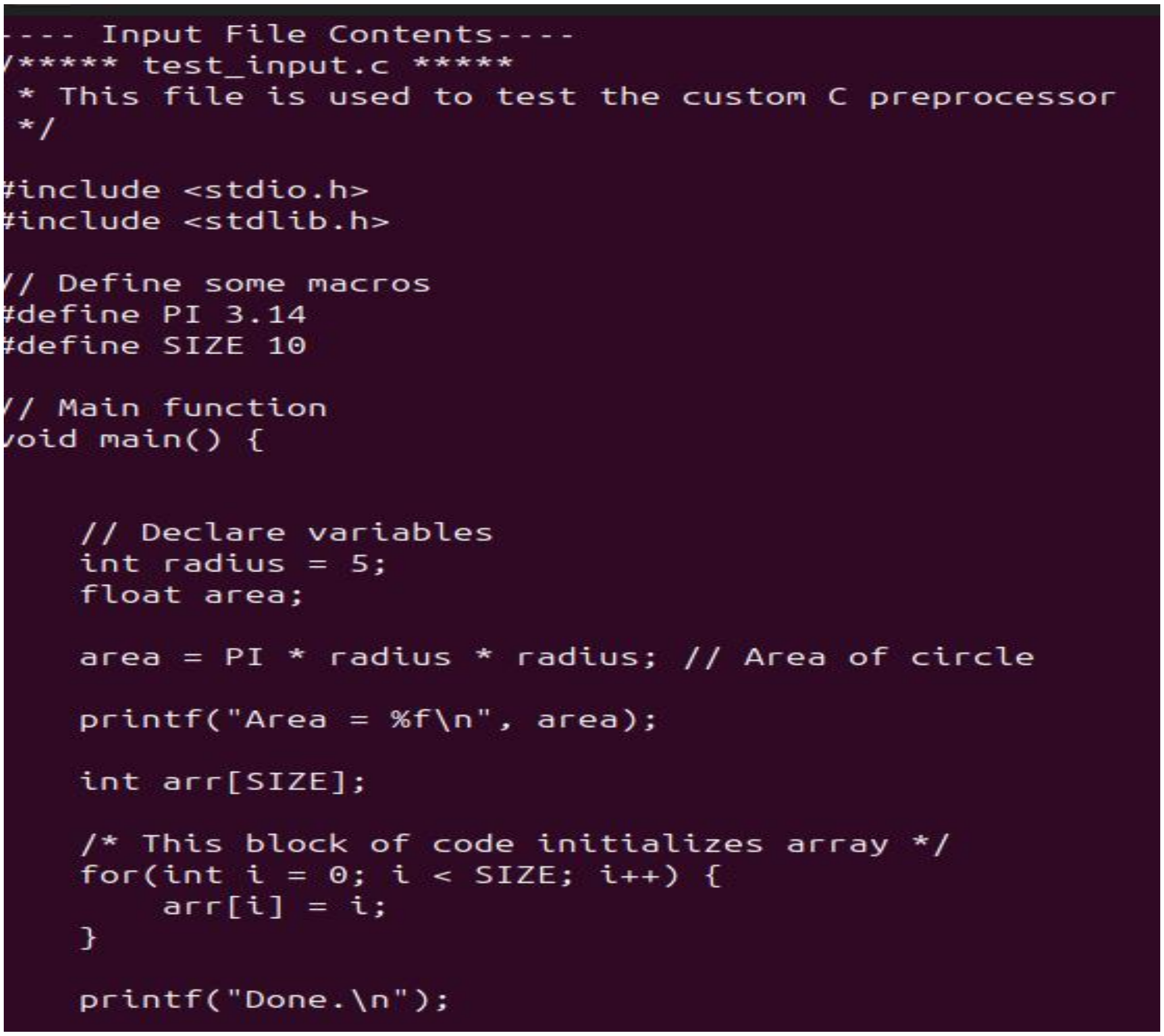


```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <input_file.c>\n", argv[0]);
        return 1;
    }

    const char *inputFile = argv[1];
    const char *noBlank = "noblank.c";
    const char *noComments = "nocomments.c";
    const char *finalOutput = "out.c";

    printf("---- Input File Contents----\n");
    displayFile(inputFile);

    removeBlankLines(inputFile, noBlank);
    removeComments(noBlank, noComments);
    macroExpansion(noComments, finalOutput);

    printf("---- Output File Contents----\n");
    displayFile(finalOutput);

    return 0;
}
```

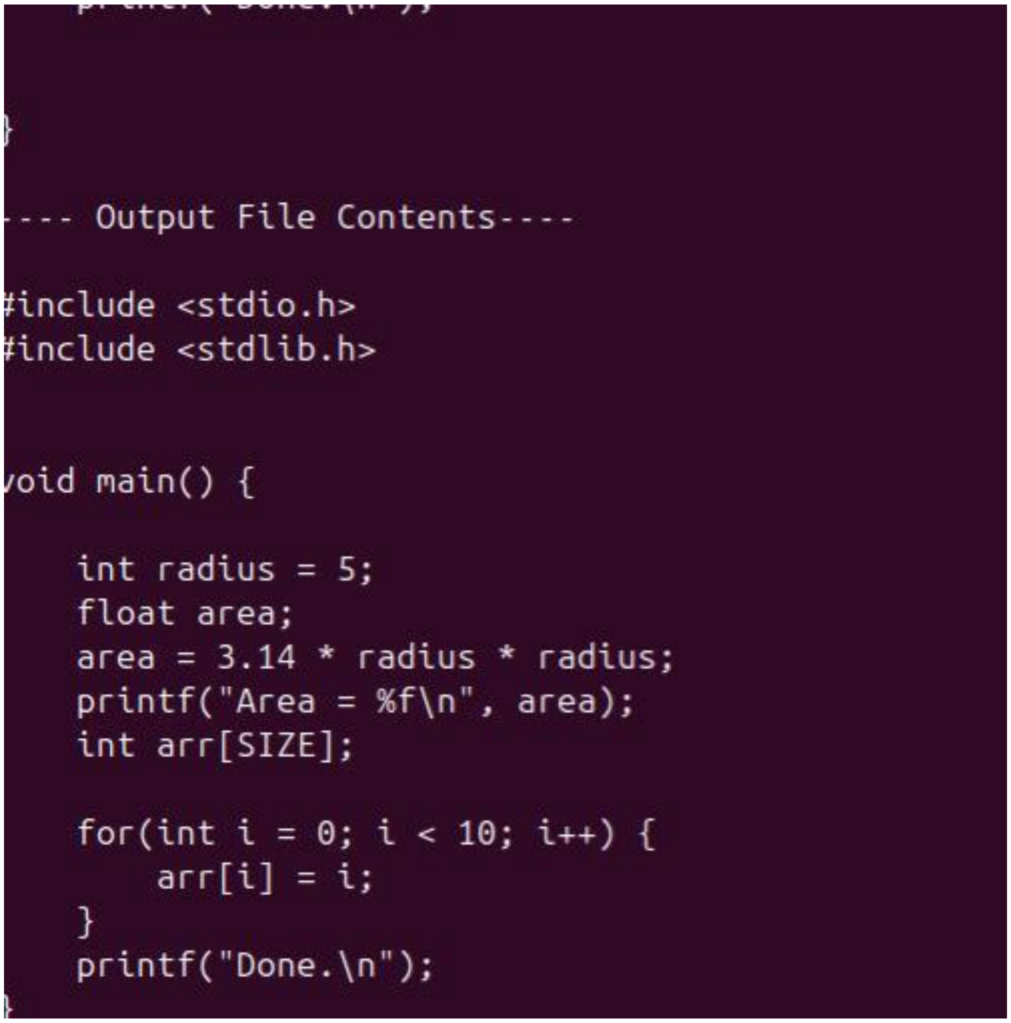
- Accepting command line arguments.
- Handling to make sure 2 arguments are passed (executable filename and input file name)
- Const Char is used as these are supposed to be preserved and not changed at all (of course filename should not change).
- Argv[1] is the name of input file.
- Noblank.c and nocomments.c are intermediary files.
- Out.c is the output file. File name are stored in approprjate variable names
- Output on console formatting
- Displaying input file
- Passing input file to remove blank line first.
- Passing noblank file to remove comments and store non commented and nonblank file in nocomments
- Passing nocomments to macroexpansion and to be stored in finalOutput file (out.c)
- Displaying out.c contents.

Outputs:

Output 1:

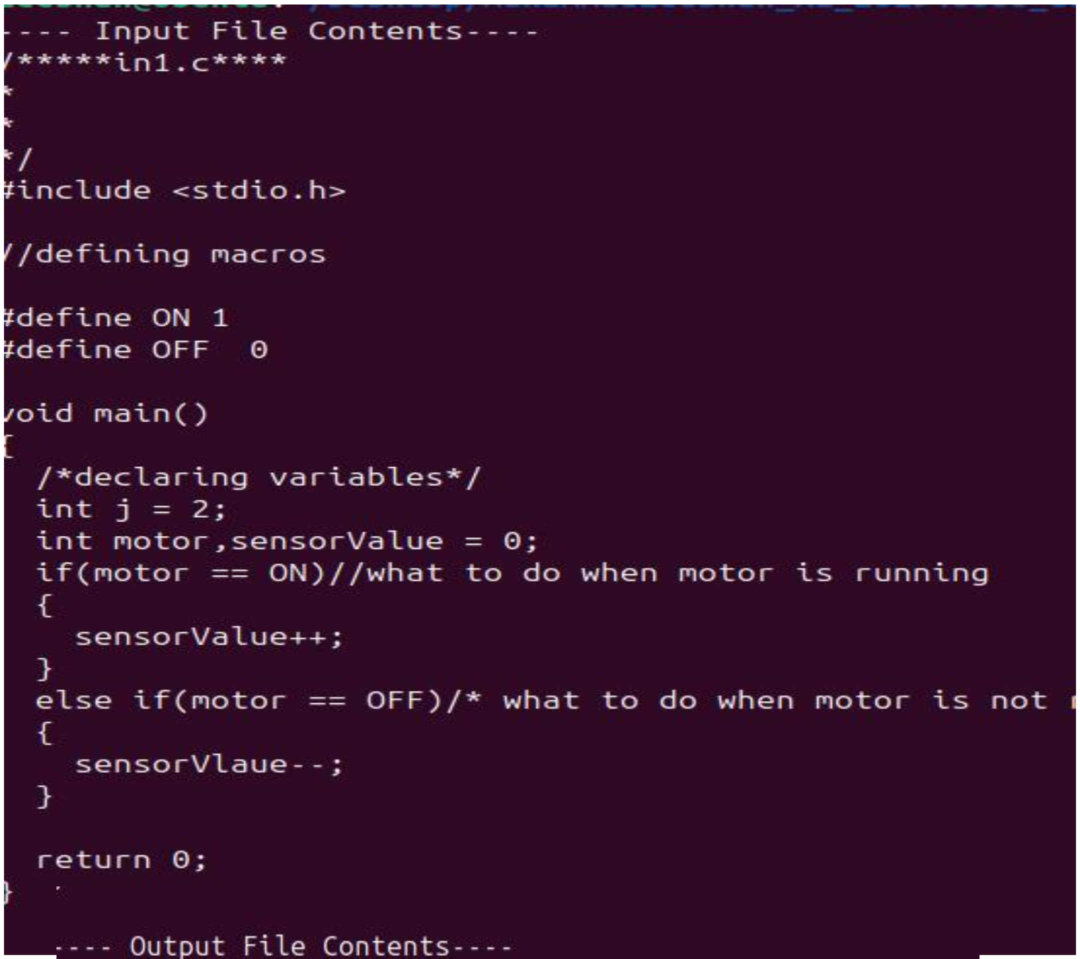


```
---- Input File Contents----
/***** test_input.c *****
 * This file is used to test the custom C preprocessor
 */

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

// Define some macros
#define PI 3.14
#define SIZE 10

// Main function
void main() {


    // Declare variables
    int radius = 5;
    float area;

    area = PI * radius * radius; // Area of circle

    printf("Area = %f\n", area);

    int arr[SIZE];

    /* This block of code initializes array */
    for(int i = 0; i < SIZE; i++) {
        arr[i] = i;
    }

    printf("Done.\n");
```

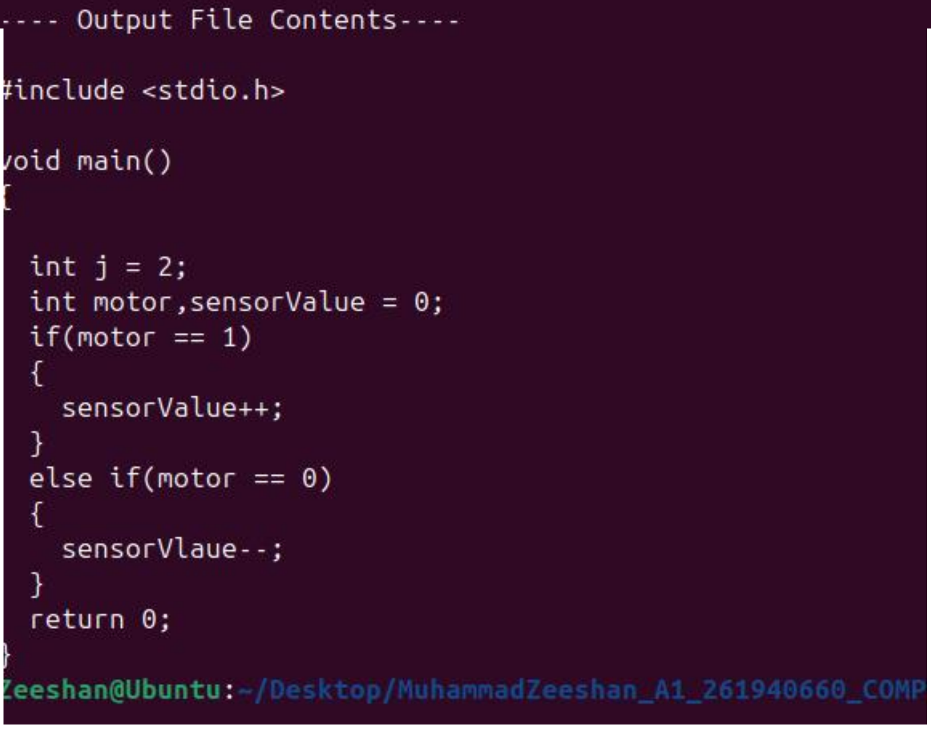


```
    printf("Done.\n");


}

---- Output File Contents----

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


void main() {

    int radius = 5;
    float area;
    area = 3.14 * radius * radius;
    printf("Area = %f\n", area);
    int arr[SIZE];

    for(int i = 0; i < 10; i++) {
        arr[i] = i;
    }
    printf("Done.\n");
}
```

Output 2:

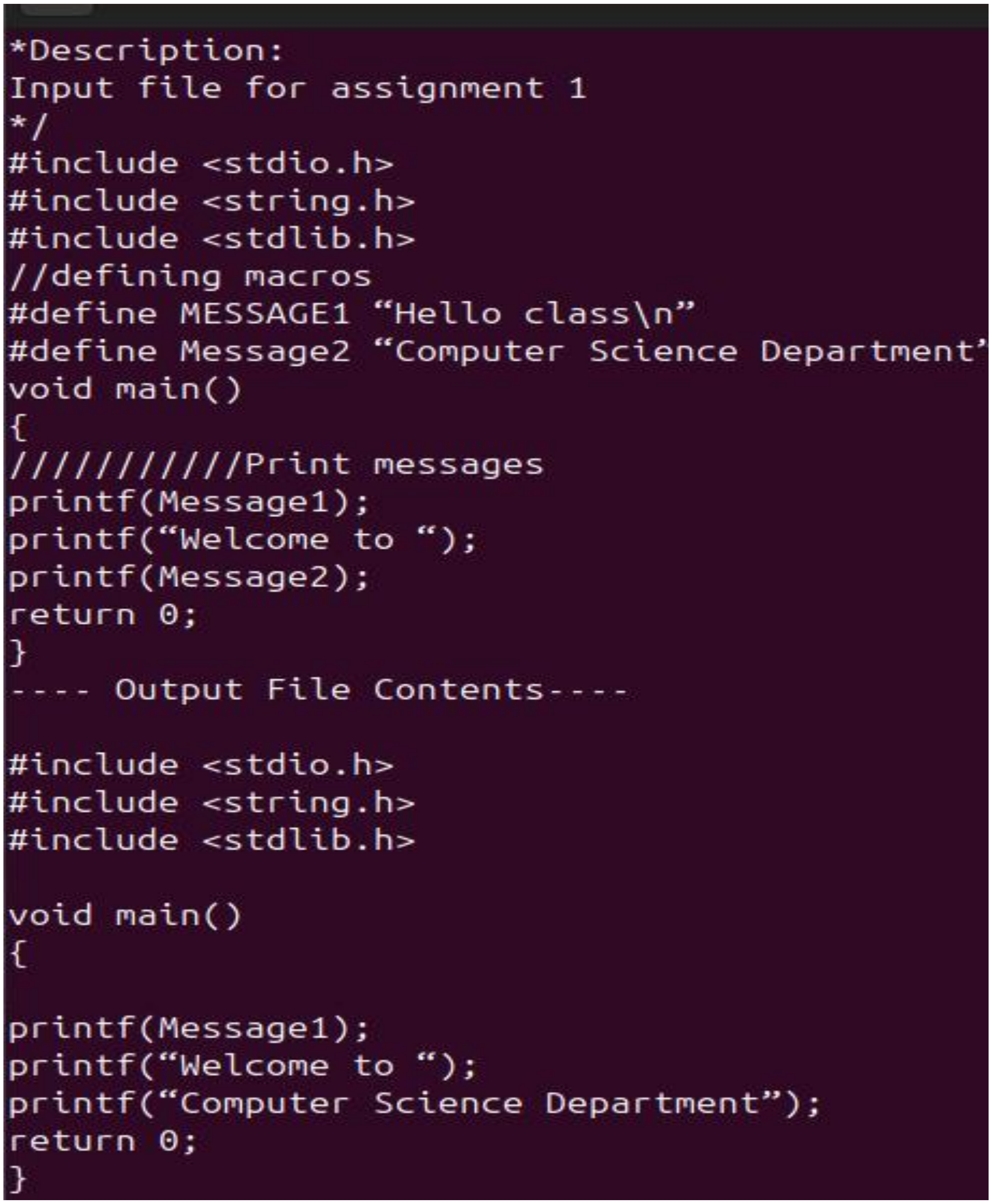


```
---- Input File Contents----
/*****in1.c****
*
*
*/
#include <stdio.h>

//defining macros

#define ON 1
#define OFF  0

void main()
{
  /*declaring variables*/
  int j = 2;
  int motor,sensorValue = 0;
  if(motor == ON)//what to do when motor is running
  {
    sensorValue++;
  }
  else if(motor == OFF)/* what to do when motor is not
  {
    sensorVlaue--;
  }

  return 0;
}
```



```
---- Output File Contents----

#include <stdio.h>

void main()
{

  int j = 2;
  int motor,sensorValue = 0;
  if(motor == 1)
  {
    sensorValue++;
  }
  else if(motor == 0)
  {
    sensorVlaue--;
  }
  return 0;
}
Zeeshan@Ubuntu:~/Desktop/MuhammadZeeshan_A1_261940660_COMP
```

Output 3:



```
*Description:
Input file for assignment 1
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
//defining macros
#define MESSAGE1 “Hello class\n”
#define Message2 “Computer Science Department’
void main()
{
///////////Print messages
printf(Message1);
printf(“Welcome to “);
printf(Message2);
return 0;
}
---- Output File Contents----

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

void main()
{

printf(Message1);
printf(“Welcome to “);
printf(“Computer Science Department”);
return 0;
}
```