P1: Preprocessor

Group GI: Adrià Soria, Alba Yerga, Roger Viader, Oriol Roca

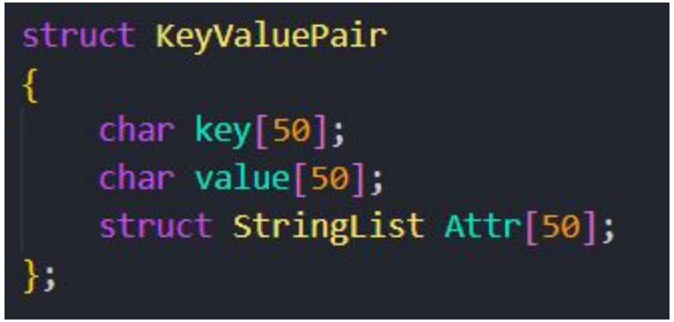
Index

System Overview

The C preprocessor is a text substitution tool that processes C source code before it is passed to the compiler. It handles directives such as #include, #define, #ifdef - #endif, parameterized macros, and comment elimination.

Data Structures Used

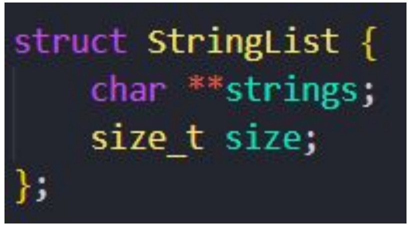
KeyValuePair



```
struct KeyValuePair
{
    char key[50];
    char value[50];
    struct StringList Attr[50];
};
```

The table to store all the values of the directives and macros. The key, when found on the code, will be substituted by the value. And for the macros, we will use the attributes.

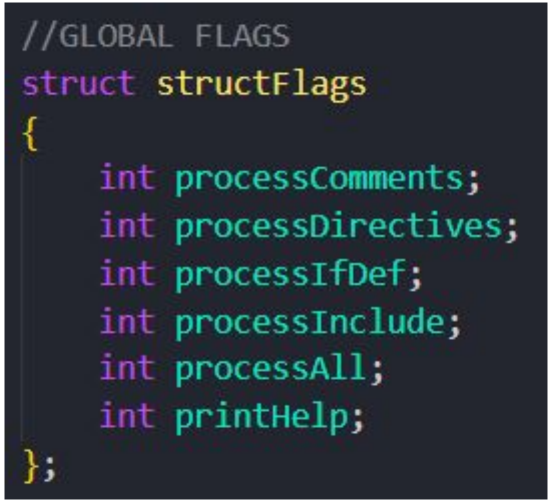
StringList



```
struct StringList {
    char **strings;
    size_t size;
};
```

The data structure to store all the attributes of the macros the preprocessor encounters throughout the code.

structFlags



```
//GLOBAL FLAGS
struct structFlags
{
    int processComments;
    int processDirectives;
    int processIfDef;
    int processInclude;
    int processAll;
    int printHelp;
};
```

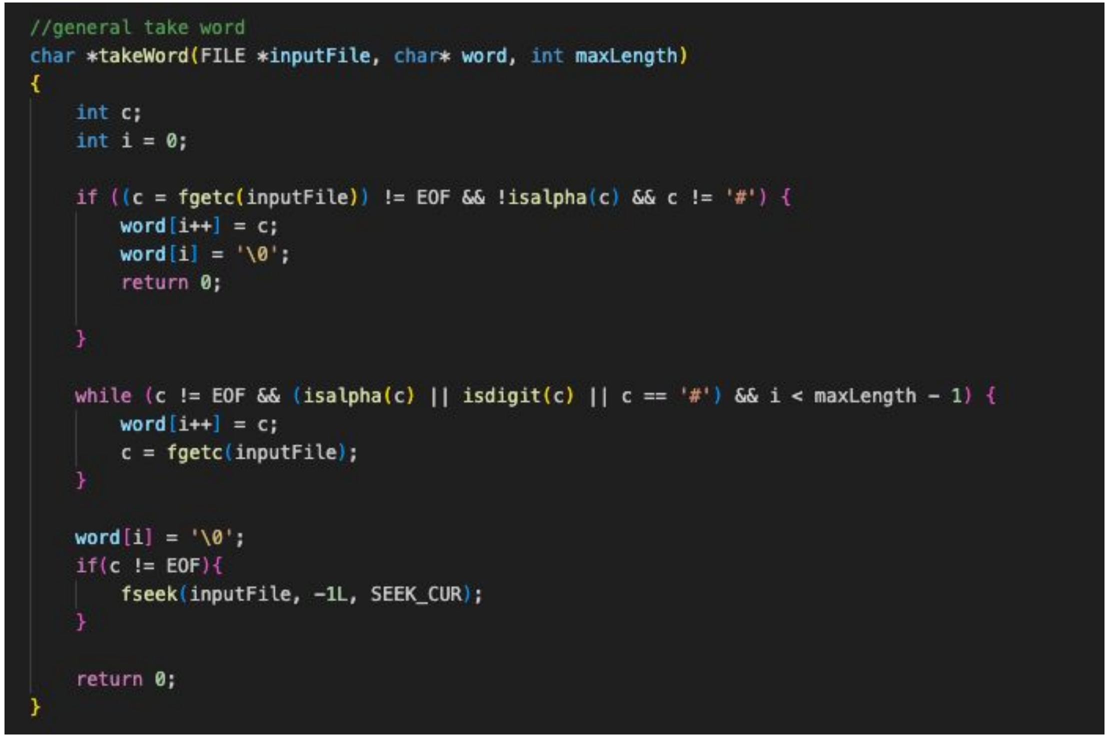
To store the flag information. It will be consulted by the preprocessor many times.

Take Word Module

This function is in charge to return the first word found on the file passed as the first argument.

That word will be saved on the word reference.

Finally, preprocessor will detect whether if the word belongs to any module and execute the functions needed.



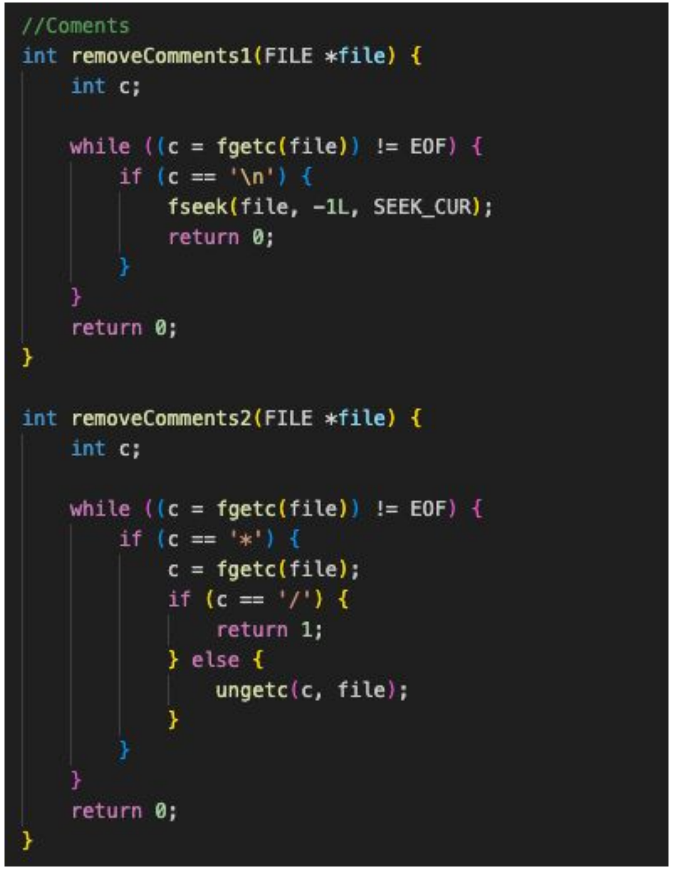
```c
//general take word
char *takeWord(FILE *inputFile, char* word, int maxLength)
{
    int c;
    int i = 0;

    if ((c = fgetc(inputFile)) != EOF && !isalpha(c) && c != '#') {
        word[i++] = c;
        word[i] = '\0';
        return 0;

    }

    while (c != EOF && (isalpha(c) || isdigit(c) || c == '#') && i < maxLength - 1) {
        word[i++] = c;
        c = fgetc(inputFile);
    }

    word[i] = '\0';
    if(c != EOF){
        fseek(inputFile, -1L, SEEK_CUR);
    }

    return 0;
}
```

Comment remove Module

For the comment remove module, we will separate one line comments and paragraph comments in two different functions.

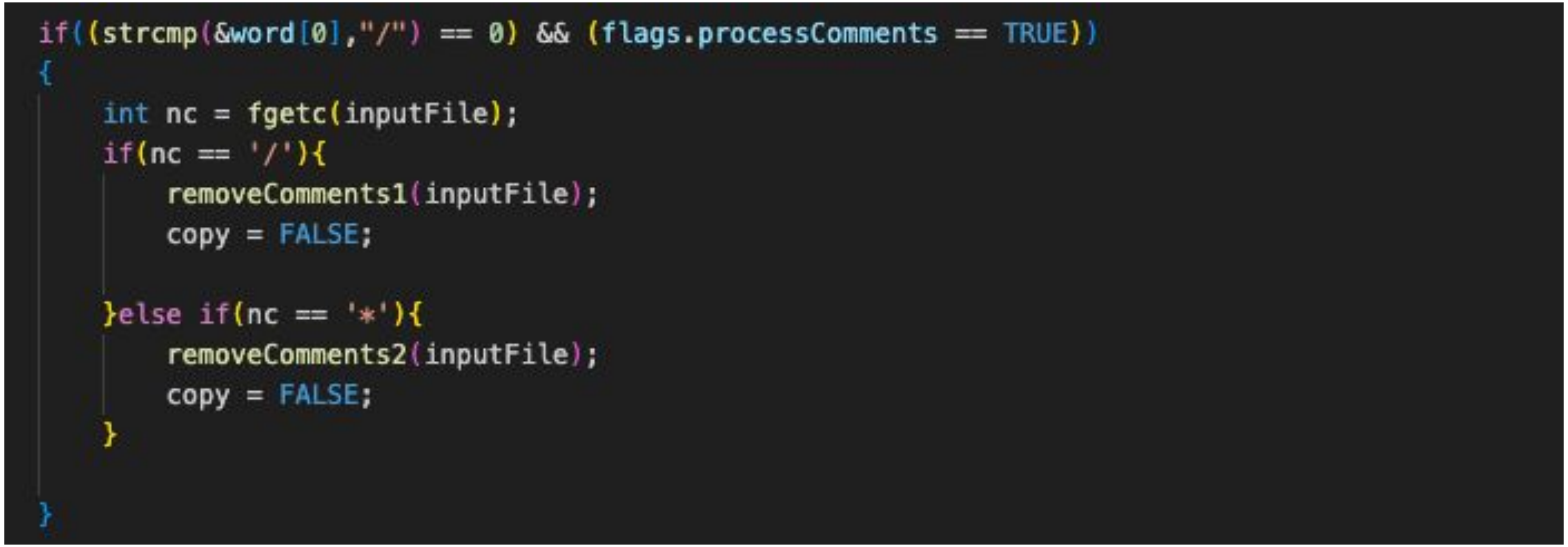
We could see that “removeComments1” works for one line comments. It will move the file pointer until it detects the ‘\n’ character. Then it stops.

Otherwise, we could see that “removeComments2” works for paragraph comments. Here, It will also move the file pointer until it detects the ‘*’ character followed by the ‘/’ character. Only in that case it stops iterating.



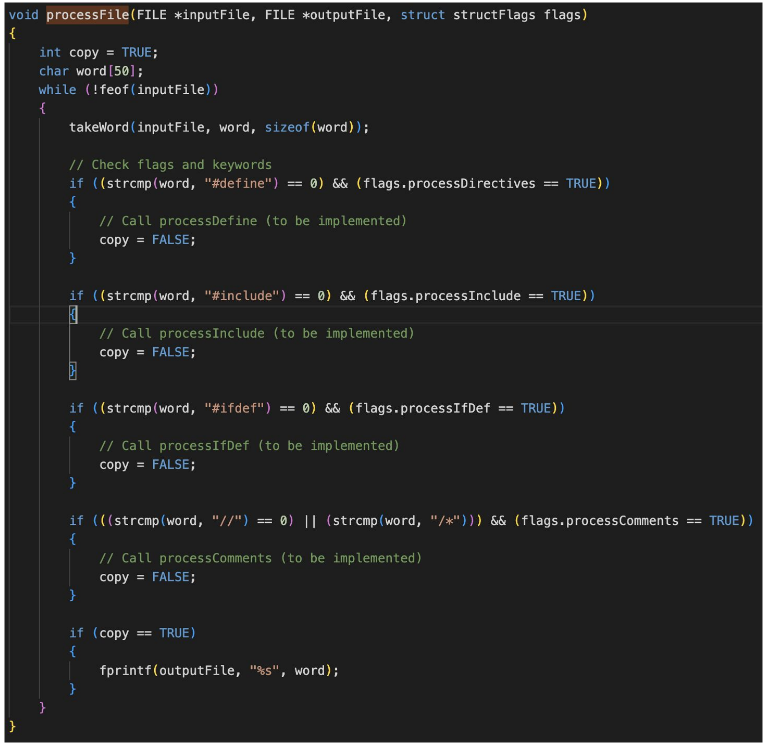
```
//Coments
int removeComments1(FILE *file) {
    int c;

    while ((c = fgetc(file)) != EOF) {
        if (c == '\n') {
            fseek(file, -1L, SEEK_CUR);
            return 0;
        }
    }
    return 0;
}

int removeComments2(FILE *file) {
    int c;

    while ((c = fgetc(file)) != EOF) {
        if (c == '*') {
            c = fgetc(file);
            if (c == '/') {
                return 1;
            } else {
                ungetc(c, file);
            }
        }
    }
    return 0;
}
```

Our code will easy decide the comment type in the following way:



```
if((strcmp(&word[0],"/") == 0) && (flags.processComments == TRUE))
{
    int nc = fgetc(inputFile);
    if(nc == '/'){
        removeComments1(inputFile);
        copy = FALSE;

    }else if(nc == '*'){
        removeComments2(inputFile);
        copy = FALSE;
    }

}
```

Process File Algorithm



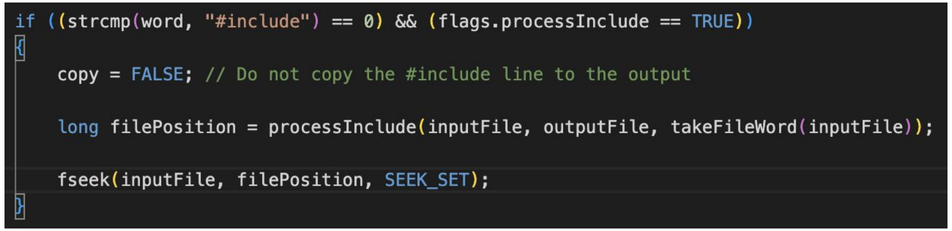
```c
void processFile(FILE *inputFile, FILE *outputFile, struct structFlags flags)
{
    int copy = TRUE;
    char word[50];
    while (!feof(inputFile))
    {
        takeWord(inputFile, word, sizeof(word));

        // Check flags and keywords
        if ((strcmp(word, "#define") == 0) && (flags.processDirectives == TRUE))
        {
            // Call processDefine (to be implemented)
            copy = FALSE;
        }

        if ((strcmp(word, "#include") == 0) && (flags.processInclude == TRUE))
        {
            // Call processInclude (to be implemented)
            copy = FALSE;
        }

        if ((strcmp(word, "#ifdef") == 0) && (flags.processIfDef == TRUE))
        {
            // Call processIfDef (to be implemented)
            copy = FALSE;
        }

        if (((strcmp(word, "//") == 0) || (strcmp(word, "/*"))) && (flags.processComments == TRUE))
        {
            // Call processComments (to be implemented)
            copy = FALSE;
        }

        if (copy == TRUE)
        {
            fprintf(outputFile, "%s", word);
        }
    }
}
```

Reads a file word by word (using **takeWord** function).

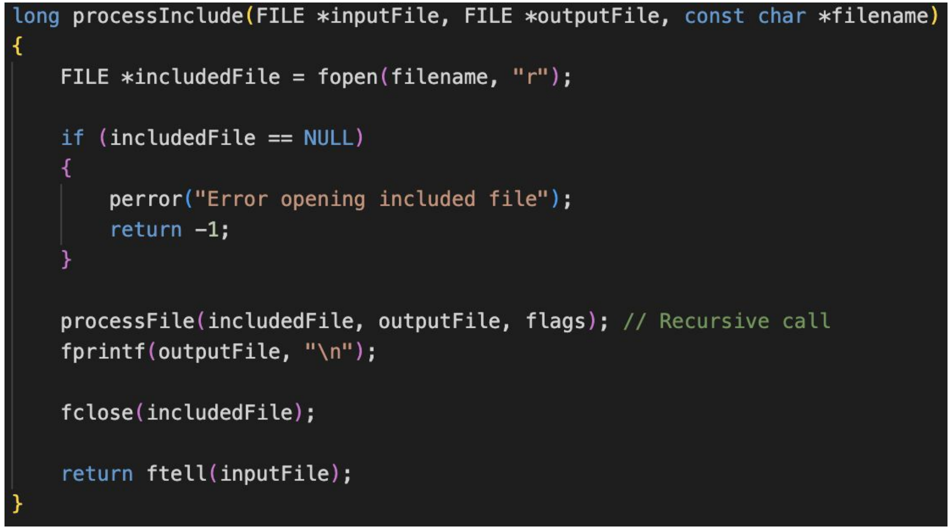
Checks if the word is a **keyword** and also if the **flag** for this keyword is **activated**.

- **If word is a keyword and the flag for the keyword is activated:** does not copy the keyword into the output file and instead, it **processes the keyword.**
- **Else: copies the word into the output file**.

Include File Algorithm



```
if ((strcmp(word, "#include") == 0) && (flags.processInclude == TRUE))
{
    copy = FALSE; // Do not copy the #include line to the output

    long filePosition = processInclude(inputFile, outputFile, takeFileWord(inputFile));

    fseek(inputFile, filePosition, SEEK_SET);
}
```



```
long processInclude(FILE *inputFile, FILE *outputFile, const char *filename)
{
    FILE *includedFile = fopen(filename, "r");

    if (includedFile == NULL)
    {
        perror("Error opening included file");
        return -1;
    }

    processFile(includedFile, outputFile, flags); // Recursive call
    fprintf(outputFile, "\n");

    fclose(includedFile);

    return ftell(inputFile);
}
```

First, we get the name of the included file with **takeFileWord**. It takes the word between "_" or between < _ > after the keyword #include.

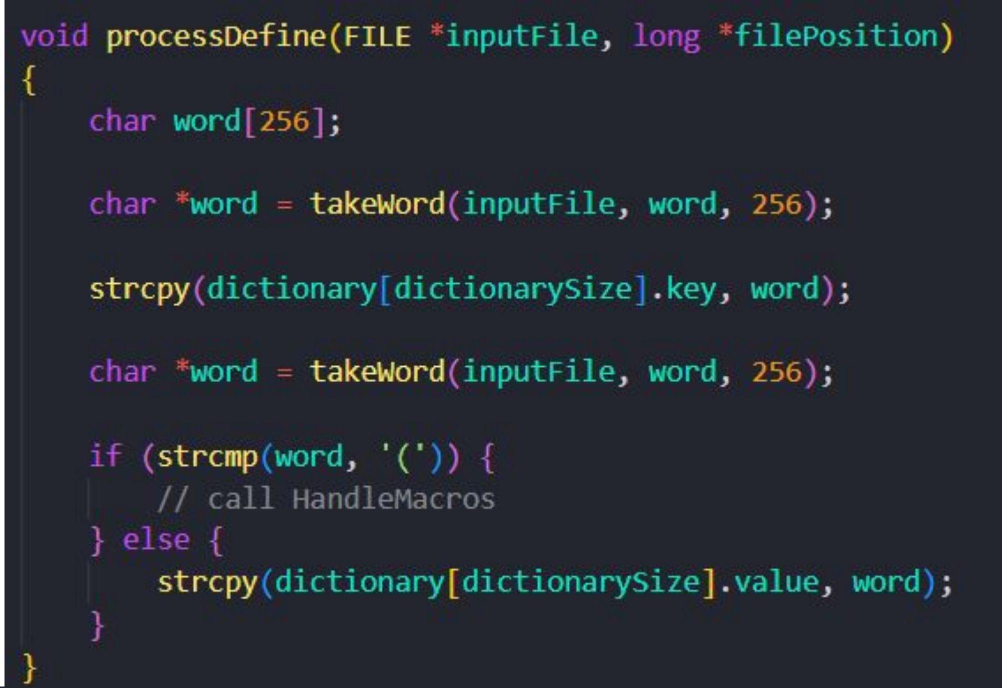
Then, we call **processInclude** that opens the included file and does a recursive call to **processFile** to process the included file.

Finally, processInclude returns a file position to keep processing.

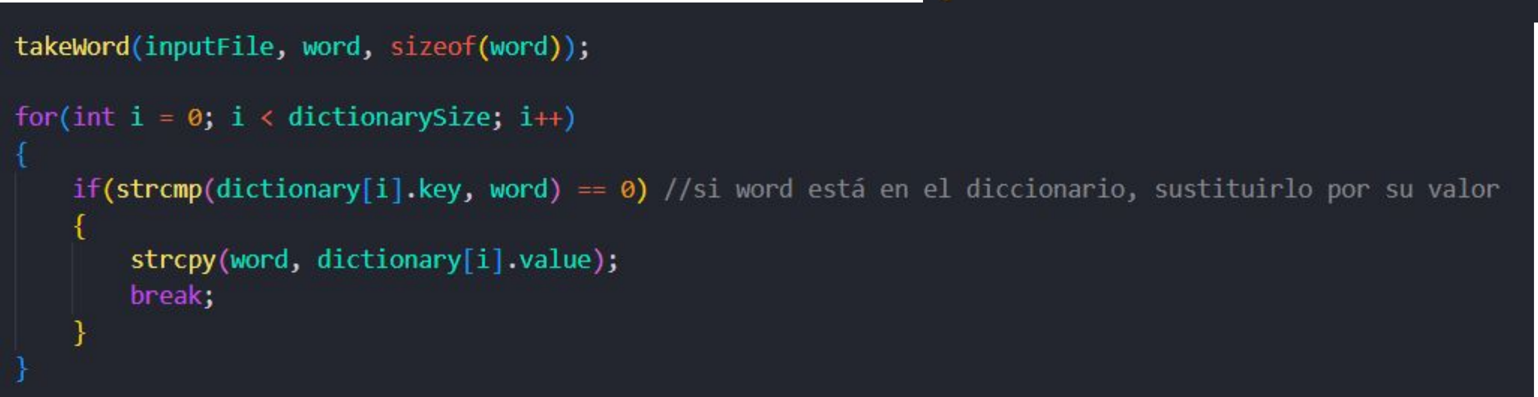
Define Substitution

The function processDefine, will store the key/value pair in the dictionary.

The for loop will be put in the while that goes through the file, when a key is found, it is replaced by the value.



```
void processDefine(FILE *inputFile, long *filePosition)
{
    char word[256];

    char *word = takeWord(inputFile, word, 256);

    strcpy(dictionary[dictionarySize].key, word);

    char *word = takeWord(inputFile, word, 256);

    if (strcmp(word, '(')) {
        // call HandleMacros
    } else {
        strcpy(dictionary[dictionarySize].value, word);
    }
}
```



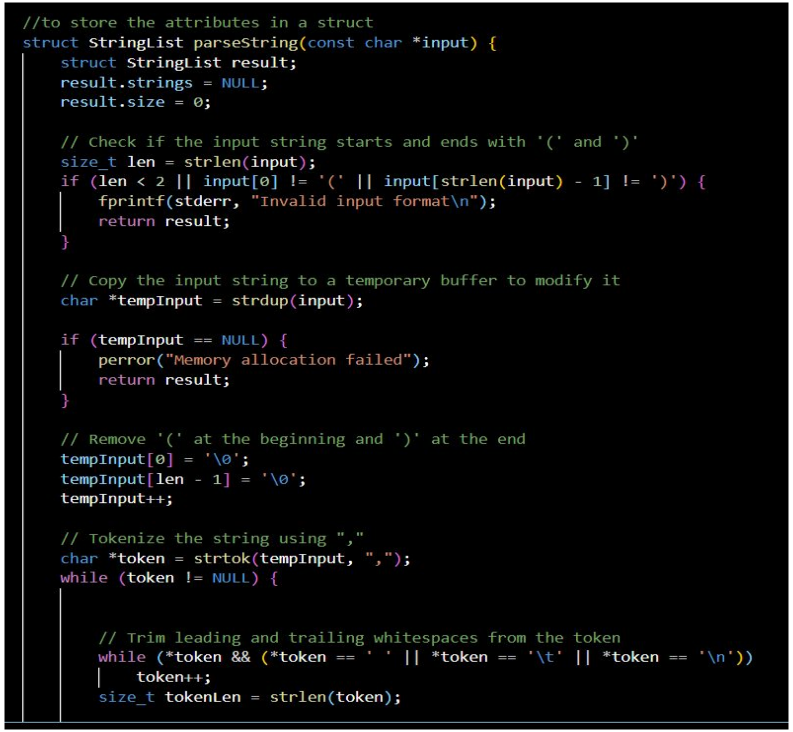
```
takeWord(inputFile, word, sizeof(word));

for(int i = 0; i < dictionarySize; i++)
{
    if(strcmp(dictionary[i].key, word) == 0) //si word está en el diccionario, sustituirlo por su valor
    {
        strcpy(word, dictionary[i].value);
        break;
    }
}
```

Macros Management

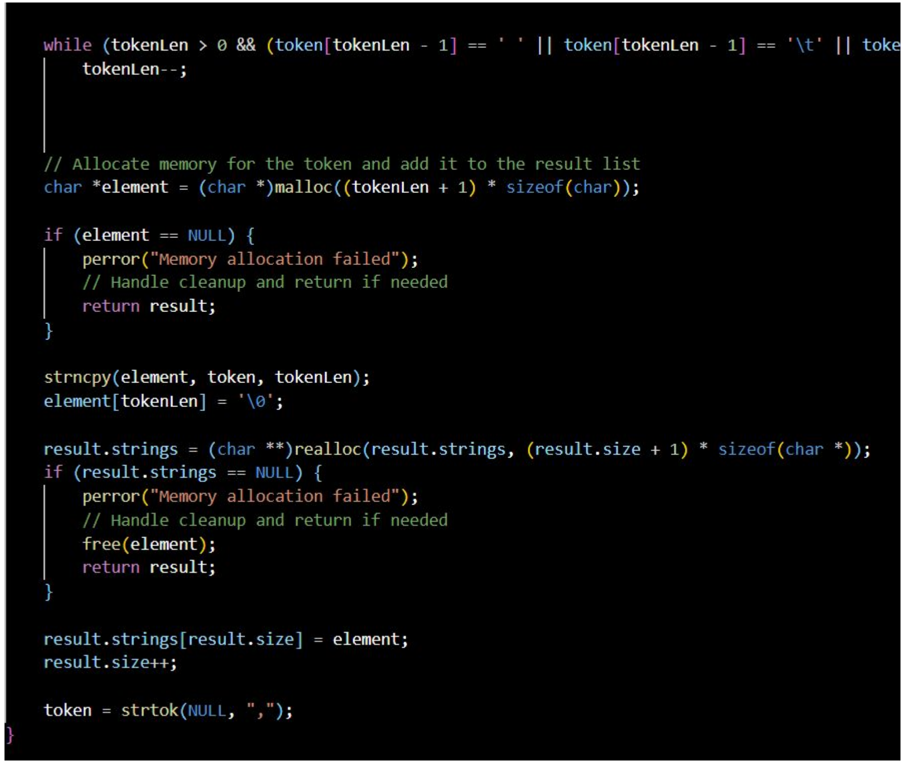
This part was not implemented because we couldn’t make it work correctly when joining it with the main code, but the idea is the following. When we encounter a #define the processMacro() function is called.

This will red the key of the macro using takeWord() and the attributes and the value using readAttributes().

readAttributes() stores the attributes in the form of a string like “(a1, a2, a3)”, so we need to store the attributes separately and tore them in a struct. We will do this using the parseString() function.

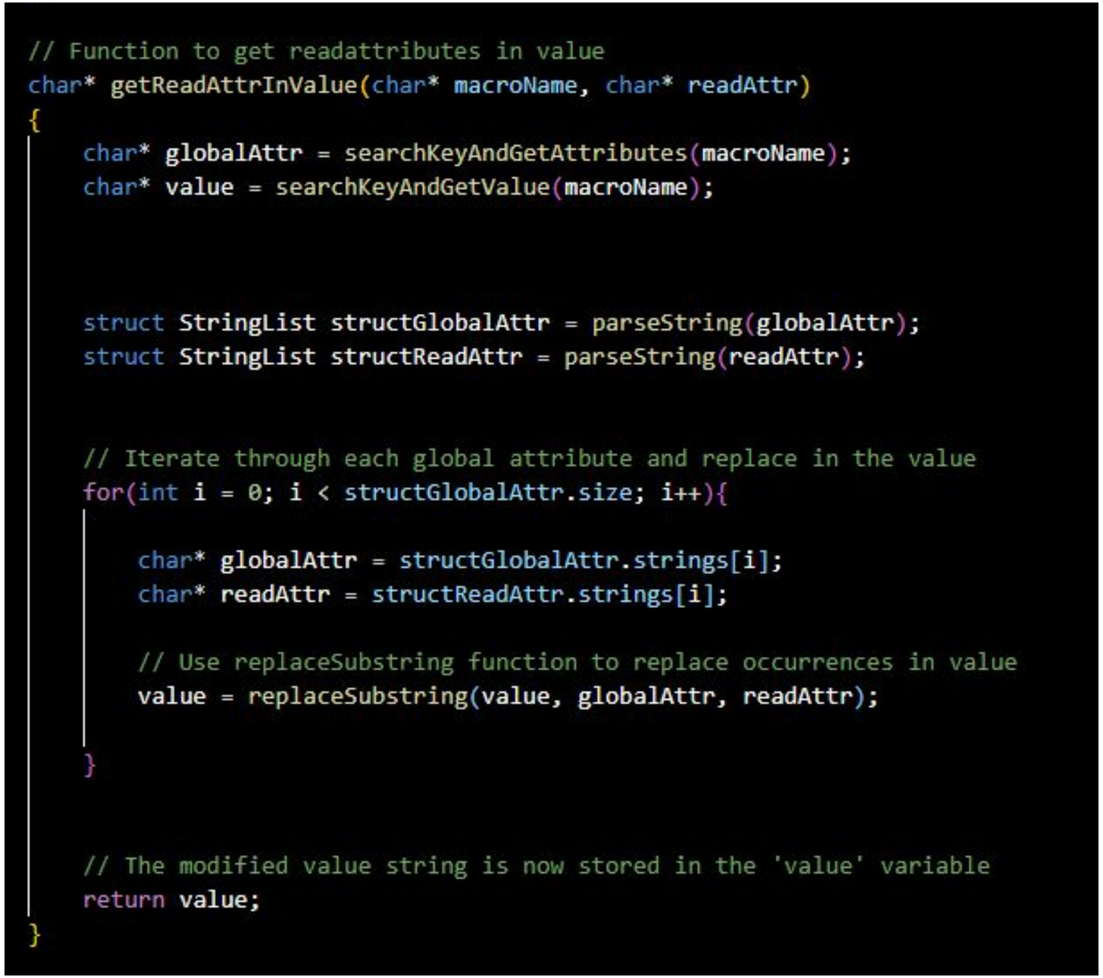


```c
//to store the attributes in a struct
struct StringList parseString(const char *input) {
    struct StringList result;
    result.strings = NULL;
    result.size = 0;

    // Check if the input string starts and ends with '(' and ')'
    size_t len = strlen(input);
    if (len < 2 || input[0] != '(' || input[strlen(input) - 1] != ')') {
        fprintf(stderr, "Invalid input format\n");
        return result;
    }

    // Copy the input string to a temporary buffer to modify it
    char *tempInput = strdup(input);

    if (tempInput == NULL) {
        perror("Memory allocation failed");
        return result;
    }

    // Remove '(' at the beginning and ')' at the end
    tempInput[0] = '\0';
    tempInput[len - 1] = '\0';
    tempInput++;

    // Tokenize the string using ","
    char *token = strtok(tempInput, ",");
    while (token != NULL) {


        // Trim leading and trailing whitespaces from the token
        while (*token && (*token == ' ' || *token == '\t' || *token == '\n'))
            token++;
        size_t tokenLen = strlen(token);
```



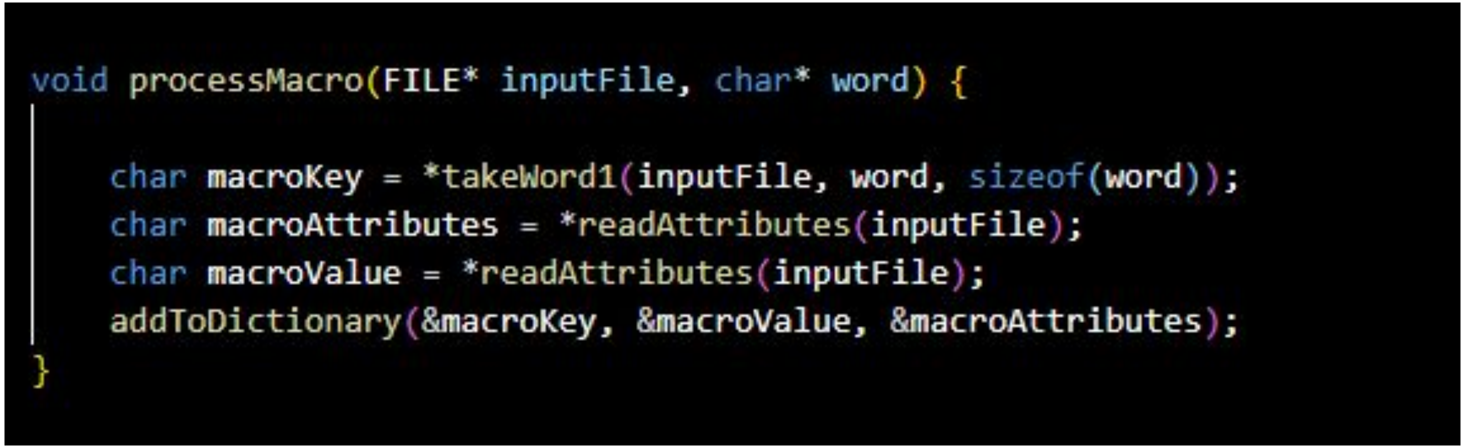
```c
    while (tokenLen > 0 && (token[tokenLen - 1] == ' ' || token[tokenLen - 1] == '\t' || toke
        tokenLen--;



    // Allocate memory for the token and add it to the result list
    char *element = (char *)malloc((tokenLen + 1) * sizeof(char));

    if (element == NULL) {
        perror("Memory allocation failed");
        // Handle cleanup and return if needed
        return result;
    }

    strncpy(element, token, tokenLen);
    element[tokenLen] = '\0';

    result.strings = (char **)realloc(result.strings, (result.size + 1) * sizeof(char *));
    if (result.strings == NULL) {
        perror("Memory allocation failed");
        // Handle cleanup and return if needed
        free(element);
        return result;
    }

    result.strings[result.size] = element;
    result.size++;

    token = strtok(NULL, ",");
}
```

Now we need a function to substitute the general attributes that are in the stored value with the read attributes from the code. To do that we use getReadAttributeInValue (), this function uses some auxiliar functions.

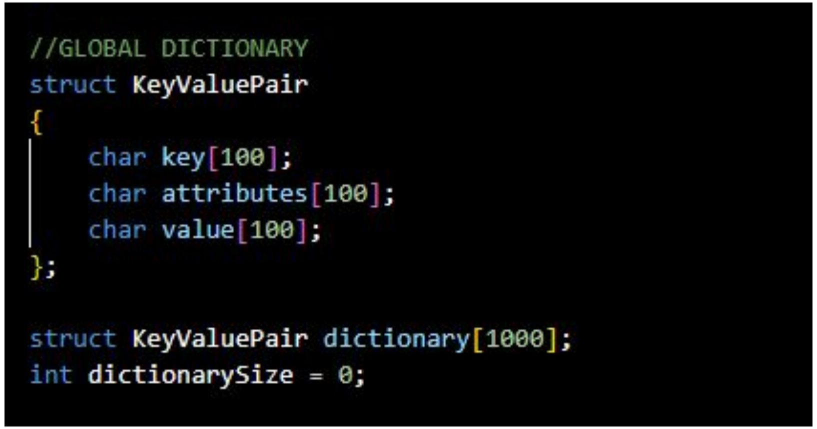


```c
// Function to get readattributes in value
char* getReadAttrInValue(char* macroName, char* readAttr)
{
    char* globalAttr = searchKeyAndGetAttributes(macroName);
    char* value = searchKeyAndGetValue(macroName);



    struct StringList structGlobalAttr = parseString(globalAttr);
    struct StringList structReadAttr = parseString(readAttr);


    // Iterate through each global attribute and replace in the value
    for(int i = 0; i < structGlobalAttr.size; i++){

        char* globalAttr = structGlobalAttr.strings[i];
        char* readAttr = structReadAttr.strings[i];

        // Use replaceSubstring function to replace occurrences in value
        value = replaceSubstring(value, globalAttr, readAttr);

    }


    // The modified value string is now stored in the 'value' variable
    return value;
}
```

Finally the function processMacro() stores the macro in the data structure.



```c
void processMacro(FILE* inputFile, char* word) {

    char macroKey = *takeWord1(inputFile, word, sizeof(word));
    char macroAttributes = *readAttributes(inputFile);
    char macroValue = *readAttributes(inputFile);
    addToDictionary(&macroKey, &macroValue, &macroAttributes);
}
```



```c
//GLOBAL DICTIONARY
struct KeyValuePair
{
    char key[100];
    char attributes[100];
    char value[100];
};

struct KeyValuePair dictionary[1000];
int dictionarySize = 0;
```