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.

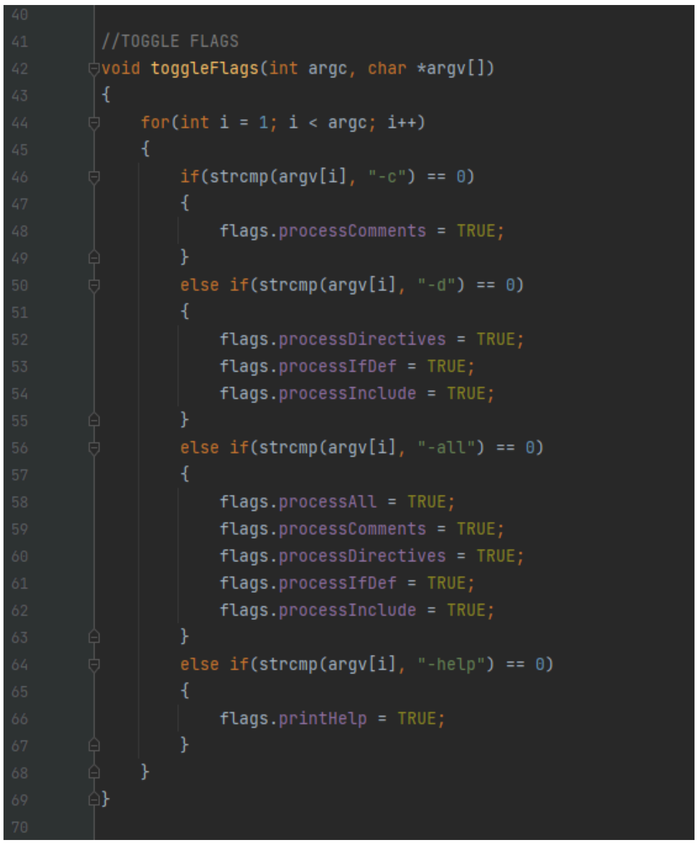
Module Breakdown

- **Main Module:** This module should be able to handle all 4 possible options.
- **Comment Handling Module:** This module should be able to eliminate all the comments.
    - Comments like: //
    - Comments like: /* */
- **Directive Processing Module:** This module should be able to handle all directives that start with the hashtag symbol (#)
    - #include → It should be substituted by the complete file that was referencing, being so that the final code is all wrapped in a single file
    - #define → With this directive, the module should replace all occurrences of the first string of the directive with the value of the following string all throughout the code
    - #ifdef - #endif → The preprocessor, as it encounters this directive, it must look at the identifier table for the string following the #ifdef directive. If the string is found in the table, then the code between #ifdef and #endif is copied to the output of the preprocessor, if not, it is taken out of the output. This way, depending on the existence of the string, the code is considered or discarded.
- **Macro Processing Module:** Is an extended process based on #define directives, the purpose is to substitute on the code the defined functions, everywhere it appears each clause. The module then substitutes all the occurrences in the code of the string named in the directive by the definition of the function coded below the directive.
    - The backlash is a continuation of the macro that allows the macro to span over more than one line. The module should also handle the backlash by eliminating it and appending all the lines in one single line of code.
- **Show “man page” Module:** This module should be able to show a manual of the preprocessor (the commands that can be used and what they do).

- **Error Handling Module:** This module should be able to handle many types of errors, also assuming that in future implementations of the code new errors will show up.
    - The errors must indicate the line of the input file where the error occurs (see __LINE__ macro in the preprocessor definition [1]).

Data Structures

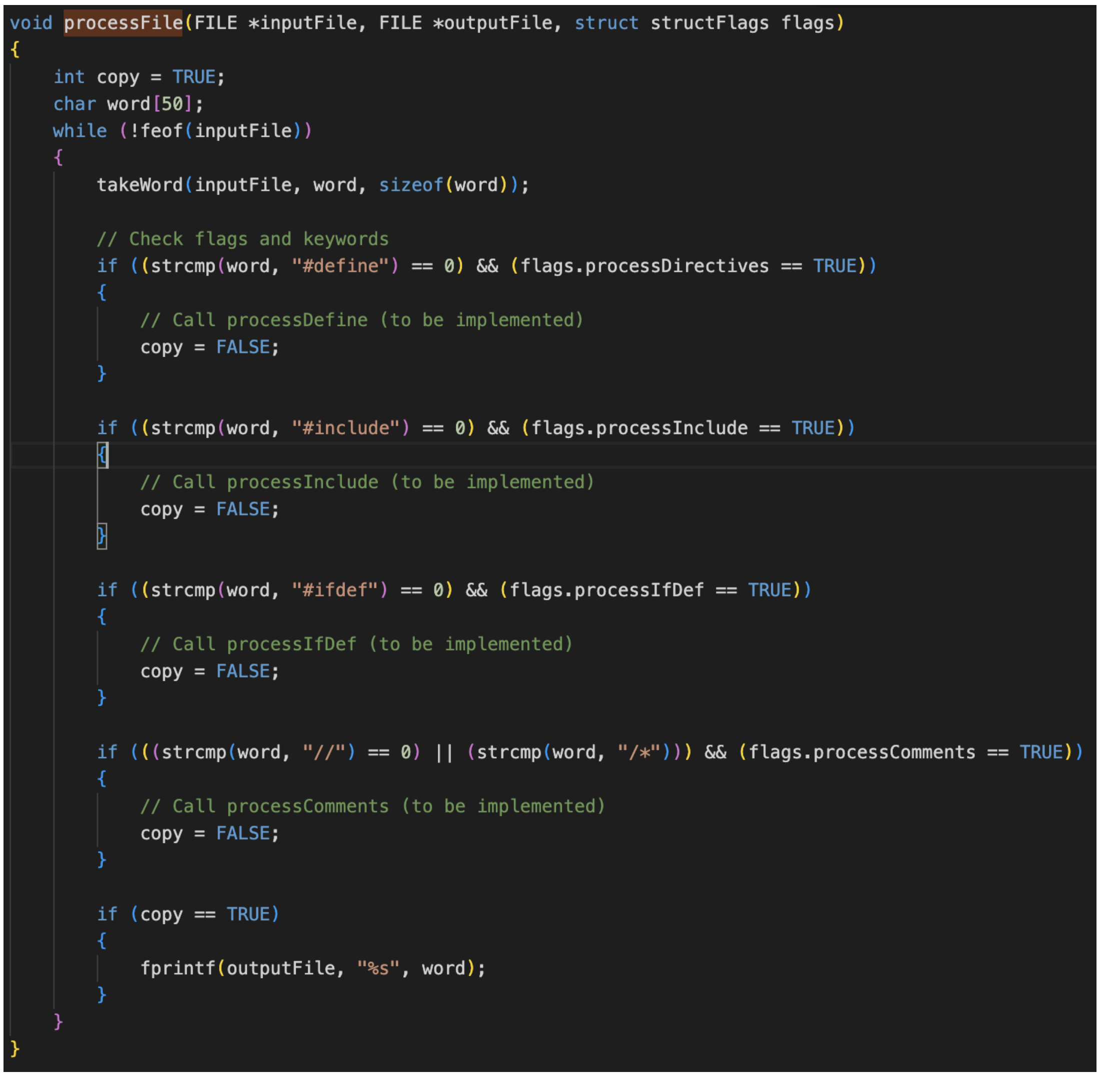
On that project we have worked with two different data structures: structFlags and keyValuePair. Data structure structFlags is composed by a group of int variables: processComments, processDirectives, processIfDef, processInclude, processAll and printHelp, each one belonging to the different run options of the preprocessor. Then function toggleFlags take charge of setting the value of each int depending on the user's preprocessor call:



```c

//TOGGLE FLAGS
void toggleFlags(int argc, char *argv[])
{
    for(int i = 1; i < argc; i++)
    {
        if(strcmp(argv[i], "-c") == 0)
        {
            flags.processComments = TRUE;
        }
        else if(strcmp(argv[i], "-d") == 0)
        {
            flags.processDirectives = TRUE;
            flags.processIfDef = TRUE;
            flags.processInclude = TRUE;
        }
        else if(strcmp(argv[i], "-all") == 0)
        {
            flags.processAll = TRUE;
            flags.processComments = TRUE;
            flags.processDirectives = TRUE;
            flags.processIfDef = TRUE;
            flags.processInclude = TRUE;
        }
        else if(strcmp(argv[i], "-help") == 0)
        {
            flags.printHelp = TRUE;
        }
    }
}

```

Algorithms

- **Take Word Algorithm:** Given a FILE*, it returns a word. The return is done when it finds a non alphanumeric or '#' character (because we need that #include, #define and #ifdef count as a word).
- **Process File Algorithm**

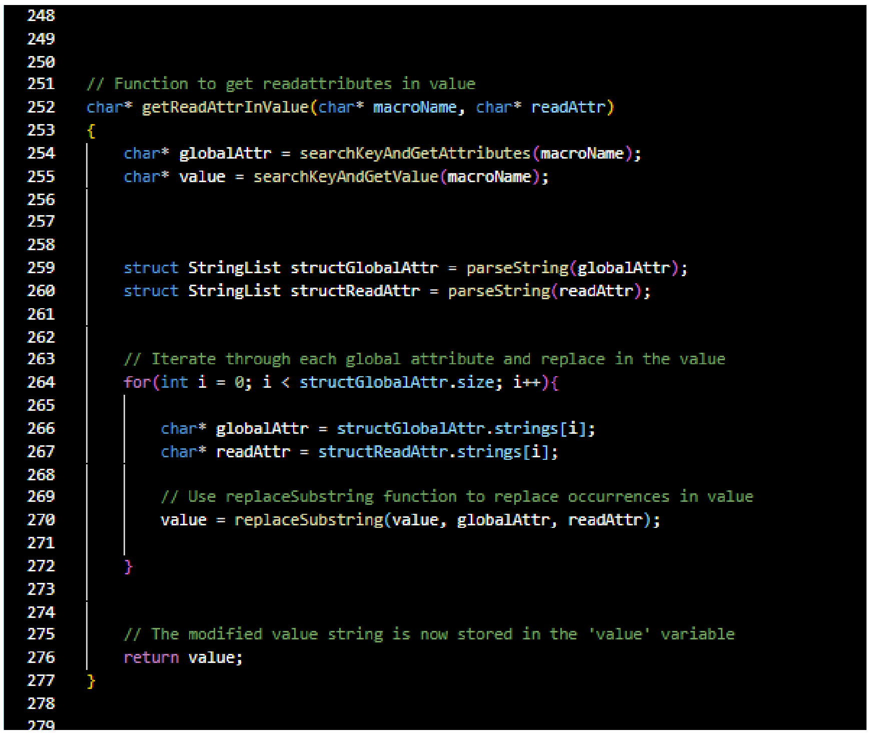


```c
void processFile(FILE *inputFile, FILE *outputFile, struct structFlags flags)
{
    int copy = TRUE;
    char word[50];
    while (!feof(inputFile))
    {
        takeWord(inputFile, word, sizeof(word));

        // 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);
        }
    }
}
```

The "processFile" function reads a file word by word (using takeWord function) and checks if the word is a keyword and also if the flag for this keyword is activated; if so, does not copy the keyword into the output file and instead, it processes the keyword. If the word is not a keyword, it just copies it into the output file.

- **Comment Removal Algorithm:** This should eliminate comments like "//…" or "/*…*/".
- **Directive Processing Algorithm**
    - **Process Include Algorithm:** This is a recursive process. After taking the filename to be included, opens that file and processes it calling again the function "processFile". In the final output file we should see the processed included files too.

   - **Process Define Algorithm:** This should store the define values and substitute them when a define key appears in the processed code.
- **Macro Processing Algorithm:** when the reading process encounters a #define the macro will be processed. We will store the macro in a structure. When encountering the macro name, we will substitute it for its value but with the specific attributes read from the file using the `getReadAttrInValue()` function.



```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;
}
```

We have used also some auxiliar functions to look for keys in the dictionary, tokenize the attributes read, replace the global attributes in the value for the local ones…
We have encountered problems when joining this part of the code with the main part, so it doesn't work correctly.

User's manual

```c
printf("MAN PAGE:\n");
printf("Usage: preprocessor [flags] <program.c>\n");
printf("Flags:\n");
printf("  -c      Eliminate comments\n");
printf("  -d      Replace all directives starting with #\n");
printf("  -all    Do all processing (comments and directives)\n");
printf("  -help   Print this man page\n");
```

Diagrams

Located inside the submission folder.
*Include **Work Distribution Diagram.***

Code File Distribution

- **comp-p1.c:** Main program file.
- **comp-p1.h:** Header file with declarations.
- **lib1.h, lib1.c:** Library files.