Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281

Project Overview

A key tool in software development is the C preprocessor, which specializes in operations such as conditional compilation, file inclusion, and macro expansion. It improves the structure of the code, makes reuse easier via macros, and makes the code easier to read by eliminating comments. It provides flexibility and customisation with features like symbolic constants and conditional compilation. All things considered, the preprocessor makes a substantial contribution to the flexibility and quality of the code prior to compilation.

Our C Preprocessor is a command-line tool developed in C that facilitates code preprocessing before compilation. Executed from the command line, it supports various flags like -c (eliminate comments), -d (replace directives), -all (combined -c and -d), and -help (display man page). The system employs pattern matching using the PatternMatcher structure for fixed and dynamic patterns, handling tasks like #define, #ifdef, #include, and comments.

Project Goals and Objectives

The creation of a reliable C preprocessor with all necessary features is the project's main objective. Our goal is to develop a flexible preprocessor that serves as a solid basis for further improvements. To keep our product accessible and flexible, we prioritize a clean code structure and simplicity of modification. We wish to provide a useful and approachable addition to the software development process by means of comprehensive testing, efficient error handling, and a well-documented user interface.

Architecture and Design

In order to improve modularity and maintainability, we have divided the codebase into separate files:

1. main.c:
    - Acts as the entry point for the program.
    - Handles command-line flags and input file processing.
    - Calls the preprocess function to transform the code.
    - Orchestrates the overall workflow of the preprocessor.

2. preprocessor.c:
    - Contains the main logic for code transformation.
    - Interfaces with the pattern_matcher and invokes specific preprocessing functions based on detected patterns.
    - Coordinates the overall preprocessing flow.

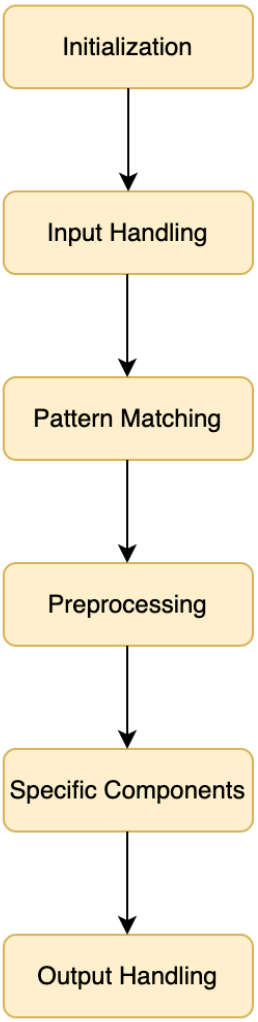
3. pattern_matcher.c:
    - Centralizes the logic for detecting specific constructs in the code.
    - Initializes and manages the pattern matcher.
    - Enables easy addition of new patterns and ensures efficient pattern matching during preprocessing.

4. utils.c:
    - Holds utility functions shared across different components.
    - Encapsulates common functionalities, promoting code reuse and reducing redundancy.
    - Enhances overall code readability and maintainability.

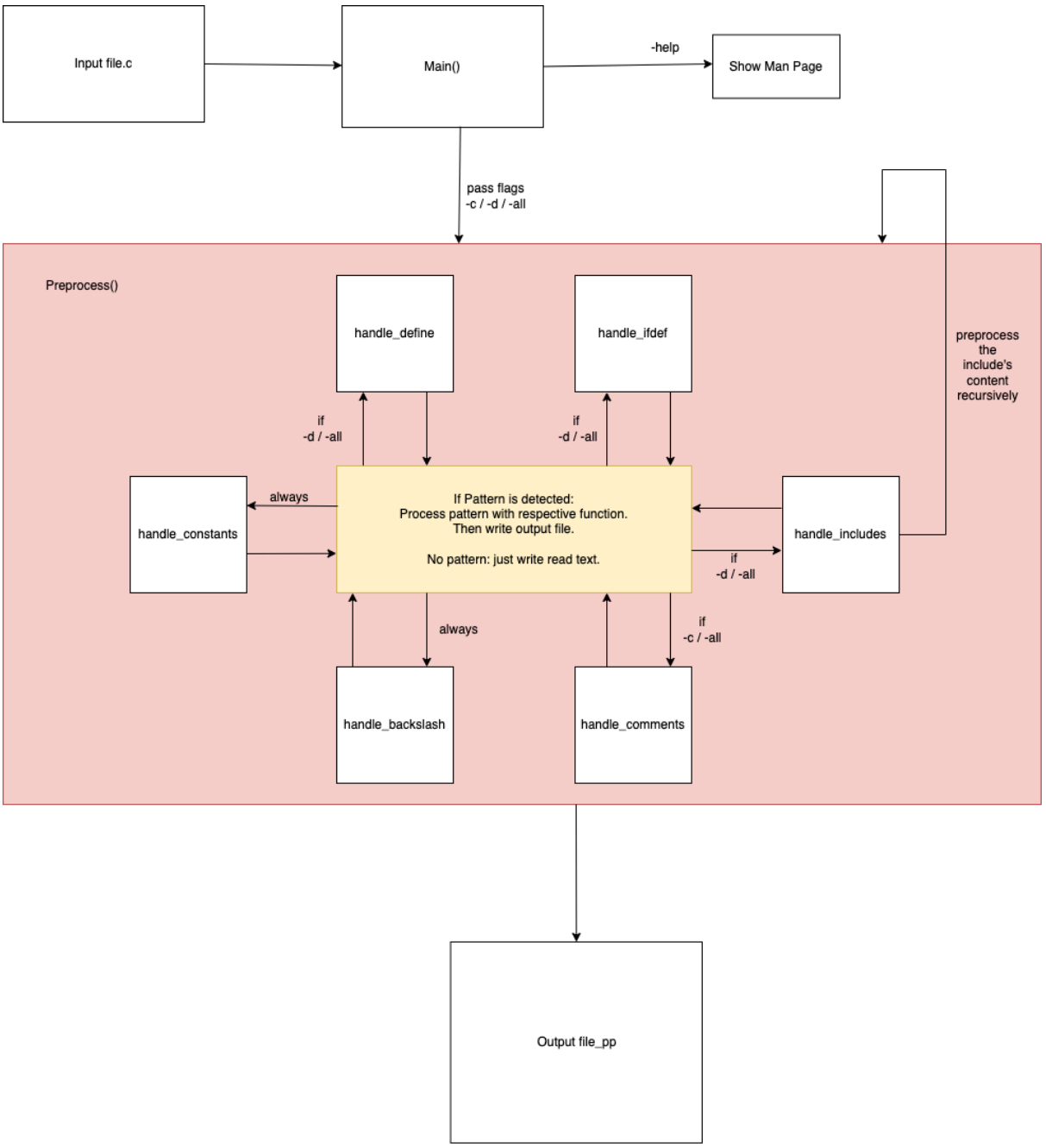
5. handle_defines.c, handle_comments.c, handle_ifdef_endif.c, etc.:
    - Dedicated files for handling specific preprocessing cases.
    - Each file contains logic tailored to its corresponding preprocessing task.

This modular file structure promotes clarity, maintainability, and collaborative development. Team members can work separately on some functionalities thanks to this distribution, which promotes teamwork during the development process.

The following diagram represents the workflow of our preprocessor:



The following diagram represents the interactions between the different components of our program:


Input file.c
Main()
-help
Show Man Page
pass flags
-c / -d / -all
Preprocess()
handle_define
handle_ifdef
preprocess
the
include's
content
recursively
if
-d / -all
if
-d / -all
handle_constants
always
If Pattern is detected:
Process pattern with respective function.
Then write output file.
No pattern: just write read text.
handle_includes
if
-d / -all
always
if
-c / -all
handle_backslash
handle_comments
Output file_pp

Functionality Descriptions

We have divided our code into the following modules:

- *handle_includes:* This function is called whenever an include directive is found in the reading buffer. Program (i.e. “#include “myheader.h””) and compiler (i.e. “#include <stdlib.h>”) file inclusion is handled by handle_include_program_files and handle_include_compiler_files, respectively. These two functions extract the given path, retrieve the file contents, apply preprocesses recursively to preprocess the contents, and then return the outcome.

  The pre_handle_compile_file and pre_handle_include_file functions are called for their respective directives during preprocessing. These functions call the relevant handle_include function, update the writing buffer with the processed material, and modify indices based on the inclusion type.

- *handle_defines:* It locates the #define block's beginning in the source code first, then goes to the character that comes after the 'e' in '#define.' It then verifies that a space exists after "#define" and indicates where the #define line ends. The length of the information that appears between "define" and the end of the line is determined.

  After that, the text is examined to see if it is a constant or a macro (contains parenthesis). Then, the defineResult function is in charge of generating a DefineInfo structure depending on arguments sent in, specifically determining the type (constant or macro) and extracting the *identifier* and *content*.

  Additionally, the file offers necessary features for a table of DefineInfo structures. These include adding new entries to the table, updating current entries, reporting the contents of the table, and determining whether an entry already exists with the same ID and identifier.

- *handle_comments:* Iterating until the end of the line or the end of the source code, the handle_comments_simple function, accepts the input source_code and starts at the designated start_index. After that, it duplicates the necessary characters,

null-terminates the result, and allocates memory for the result string—which does not include the single-line comment. The function modifies the position after the processed comment in the new_index variable.

Multi-line comments are handled similarly by the handle_comments_multi function. Until it discovers the closing "/" or reaches the end of the source code, it begins at the first "/". The function modifies new_index in accordance with the memory allocated for the outcome, which does not include the multi-line comment.

- *handle_ifdef_endif:* It takes a given #ifdef structure index, isolates it, and returns code blocks related to it. The function iterates over the source code, locating the desired #ifdef directive with the help of a counter (ifdef_index). Once found, it locates the matching #endif and extracts the code block inside for additional preparation.

  Before using handle_ifdef_endif, the preprocessing procedures, which are contained in pre_handle_ifdef_endif, entail deleting single-line (//) and multi-line (/* */) comments from the reading buffer. The code block linked to the designated #ifdef index is represented by the result, which is appended to the writing buffer.

- handle_backslash: It takes a given #define, we search if the macro has a / between characters from the string. When we detect a backslash at the end of the line, we delete it and we join the lines.

Command-Line Flags

The program provides various options to control the level of preprocessing through command-line flags. Here's an explanation of each flag:

1. -c (Comments):
   - Behavior: When specified, the program removes comments from the code.

2. -d (Directives):
   - Behavior: Replaces all directives starting with '#'.

- Example: If the code contains #define MAX 100, applying the -d flag will replace this directive with its processed form.

3. -all:
    - Behavior: Performs all implemented preprocessing. Equivalent to combining -c and -d.
    - Examples:
      -all processes both comments and directives.
      -c -d -all is correct syntax and results in the same behavior as -all.
4. -help:
    - Behavior: Prints a "man page" or help information about the preprocessor.
    - Note: If -help is included, it neglects any other flags and only prints the help page. No actual preprocessing is performed.

Order of Flags: The order of flags is not relevant. For instance, -c -d is equivalent to -d -c. Multiple flags can be combined, and they will be interpreted as a request for the combined functionalities.

Default Behavior: If no flags are provided, the program assumes the default behavior, which is to eliminate comments.

Help Page: The help page or "man page" provides user information about the preprocessor, its options, and how to use them. To view the help page, use the -help flag. It takes precedence over other flags and doesn't execute preprocessing.

Testing Approach

We have designed a wide range of test cases that address both common and difficult circumstances in order to verify the preprocessor's functionality. These tests evaluate a number of factors, including the way it manages instructions, comments, and inclusions. To ensure that each flag and preprocessing feature is proper and produces the desired results, we have concentrated on functional testing.

Individual Responsibilities

Breakdown of the tasks and roles assumed by each member of the team:

- *Handle includes*: Marcel Aranich and Arnau Solans
- *Handle ifdef and endif*: Ariadna Prat
- *Handle defines (constants and macros)*: David Garcia and Clàudia Quera
- *Handle Comments*: Jorge Villarino