**RELATÓRIO – TRABALHO I**

**COMPRESSOR PARA ARQUIVOS DE TEXTO**

Matheus Percário Bruder, Arthur Guedes de Souza, Gabriel Velasco Fernandes, Kevin Barrios

**INTRODUÇAO**

O trabalho fará parte do processo de avaliação da disciplina (ST562) Estruturas de Arquivos, da Faculdade de Tecnologia da UNICAMP de Limeira.

O objetivo do trabalho foi implementar um compressor para arquivos de texto, contudo, haviam dois casos de uso. O primeiro caso de uso (C1) consistia na implementação de um compactador, isto é, a partir de um arquivo de texto qualquer, o programa deveria ler, no máximo, blocos de 4096 *Bytes* e então substituir as palavras com mais de três caracteres por 2 *Bytes*. Já o segundo caso de uso (C2) consistia na implementação de um descompactador com base no arquivo que havia sido compactador anteriormente no C1.

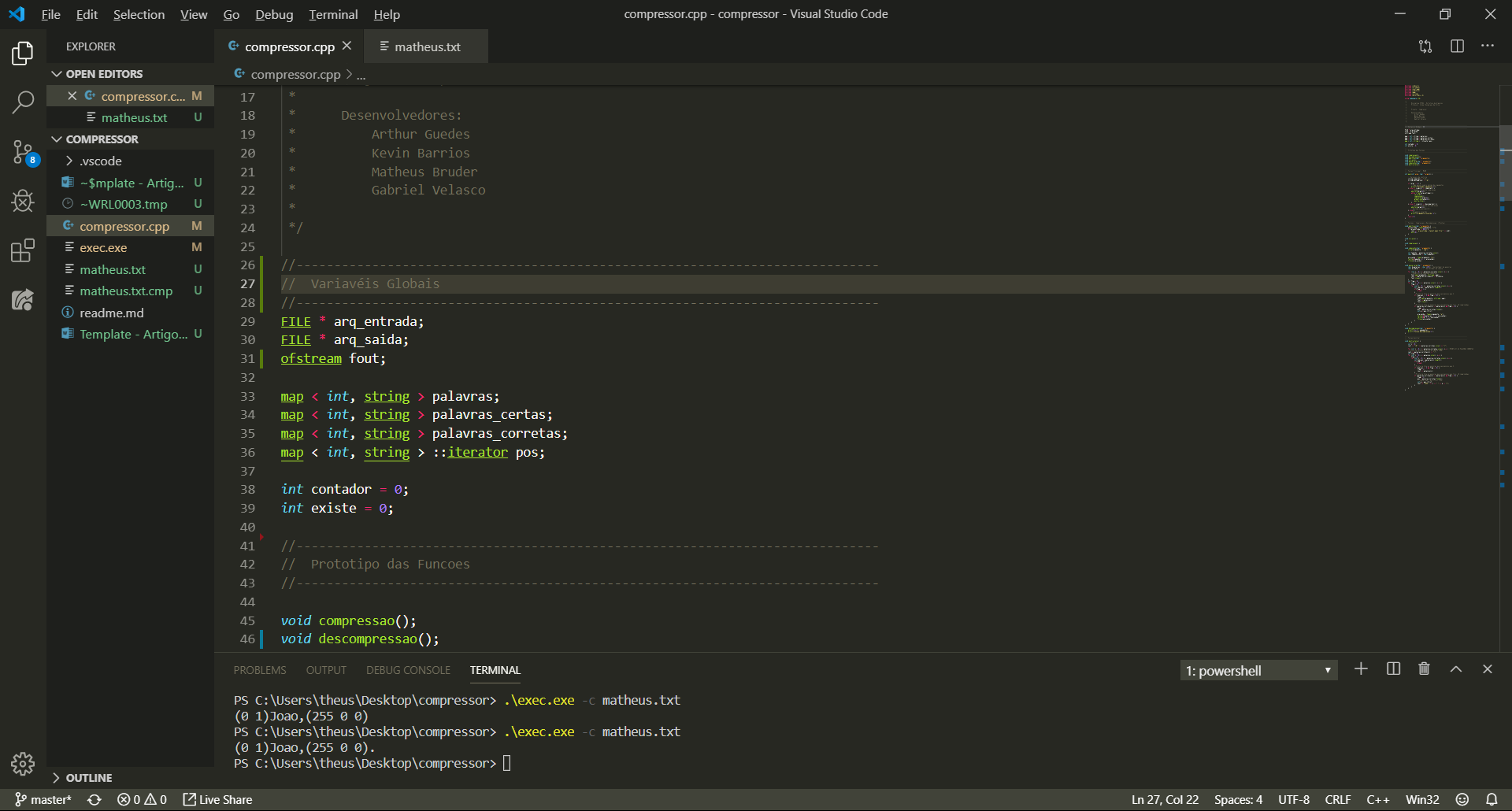
O trabalho tinha algumas limitações, como por exemplo, não há tratamento entre letras maiúsculas e minúsculas, isto é, as palavras “Joao” e “joao” são entendidas como palavras diferentes, logo, possuirão códigos diferentes. Além disso, o código não trata palavras com acentuação, portanto, deve-se realizar um tratamento para textos escritos em português antes de submeter ao compressor.

Para utilizar o programa, basta compila-lo e então utiliza-lo via linha de comando, tanto no prompt do Windows como no terminal do Linux. Para isso, os comandos que devem ser informados, são: “***nome\_executável -c nome\_arq.txt***”. Ao executar o comando anterior, será gerado um arquivo compactado com o seguinte nome: “***nome\_arq.txt.cmp***”, em que “***.cmp***” é um padrão requisitado pelo professor para informar que o arquivo em questão está compactado.

Dada essa sucinta introdução, sobre o trabalho I, o grupo conseguiu implementar apenas o C1, ou seja, esse trabalho atendeu somente metade das especificações propostas pelo professor.

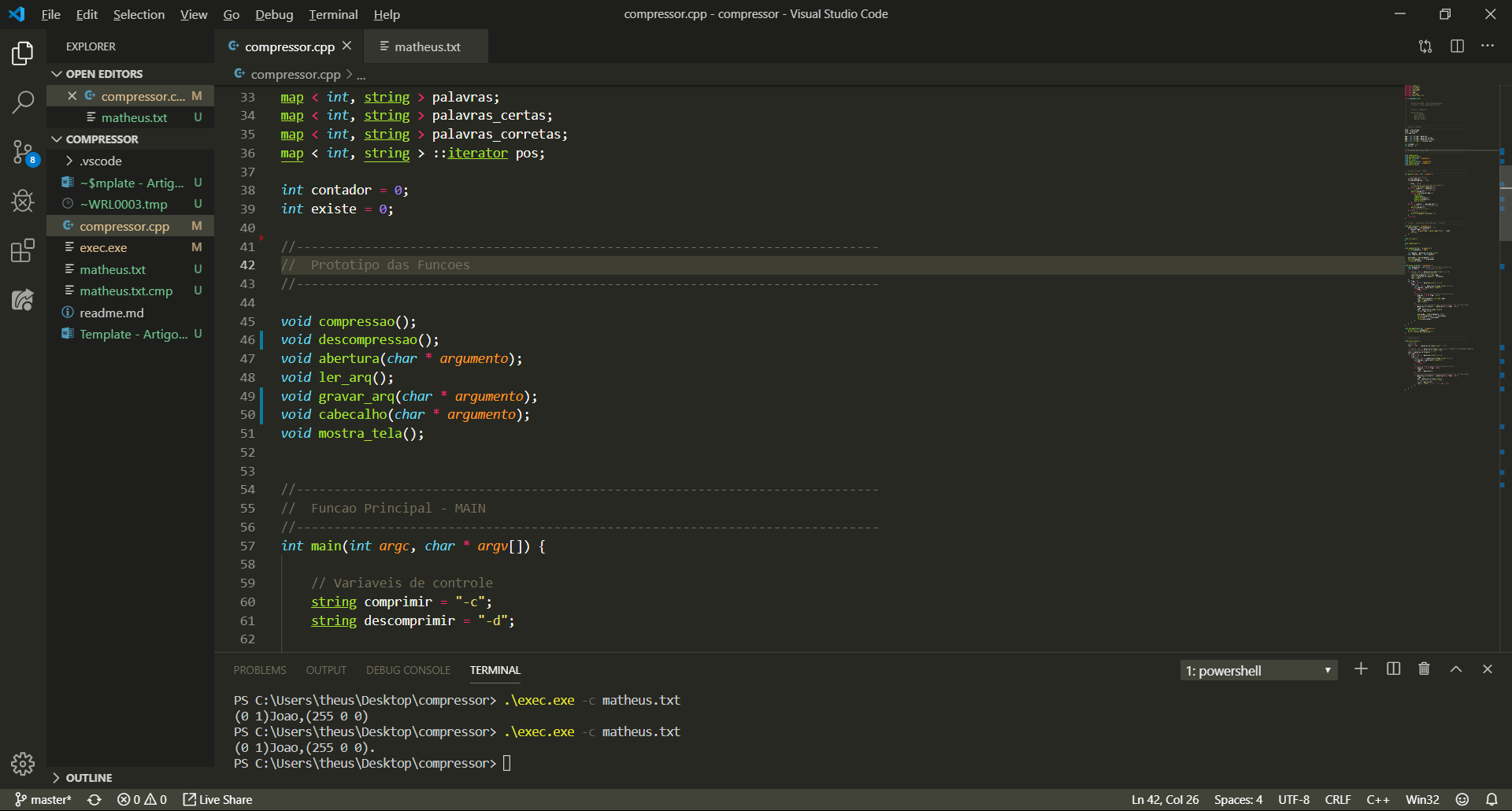
**DESCRIÇÃO DE ATIVIDADES**

O compressor foi implementado pelo grupo utilizando a linguagem de programação **C++** e as bibliotecas utilizadas foram as seguintes: ***<stdio.h>, <string.h>, <iostream>, <stdlib.h>, <map>, <string> e <bits/stdc++.h>***. Além disso, o compactador possuía algumas variáveis globais, entre elas estão dois ponteiros, um para arquivo de entrada que deve ser lido e outro para um arquivo de saída que deve gravar o texto compactado. Além disso, há também três *maps*, os quais servirão para tratar as palavras lidas do texto original, um *iterator* e duas variáveis de controle do tipo *int.*



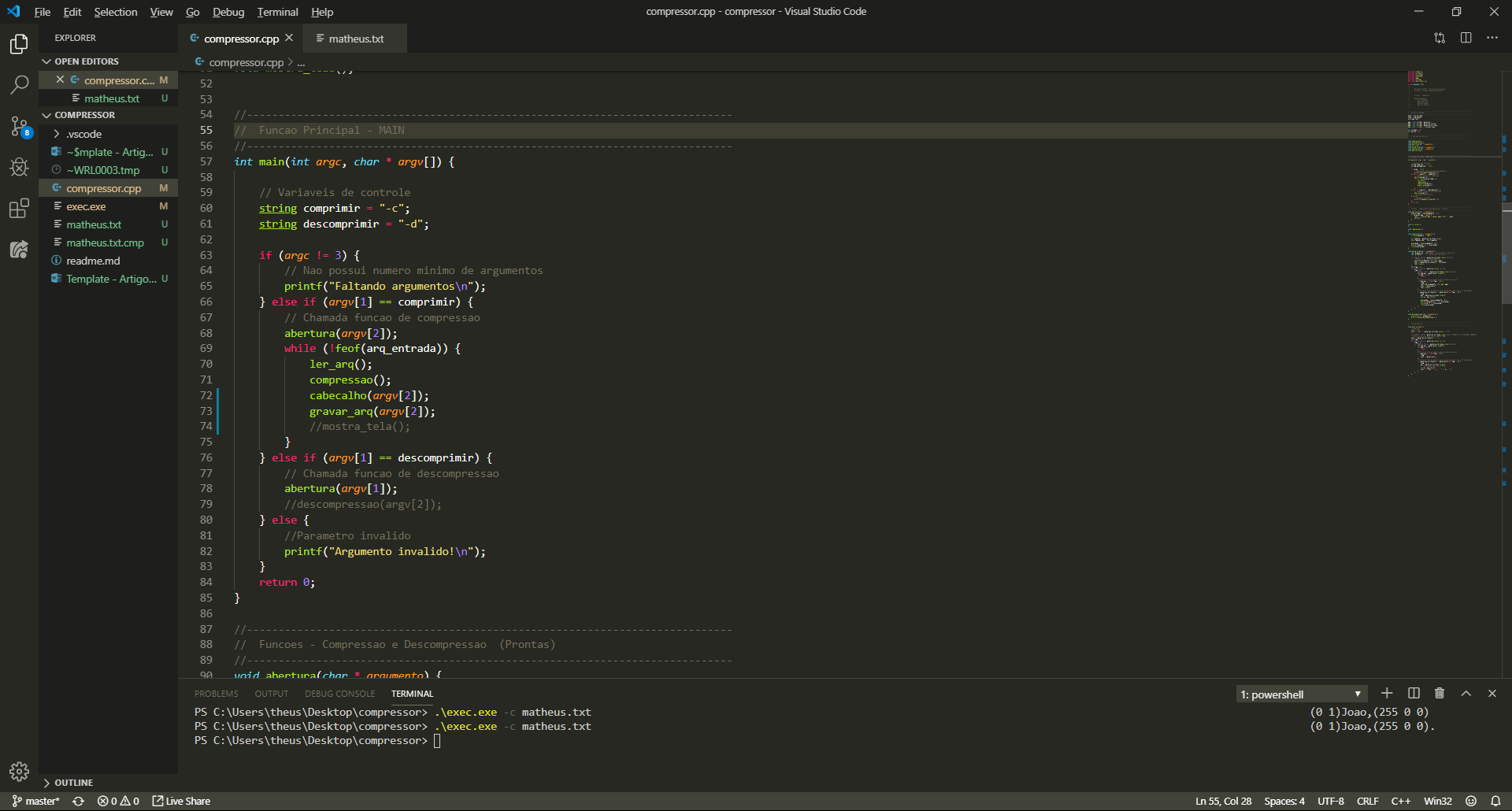
***Figura 1*** *– Variáveis globais do programa*

Além das variáveis globais exemplificadas na Figura 1, o programa também possui o protótipo de cada uma das funções necessárias, isto é, o programa está totalmente modularizado.



***Figura 2*** *– Protótipos das funções do programa*

A função principal, função main, por sua vez, faz a chamada de cada uma das funções no período especifico. Contudo, antes de chamar as funções há a verificação para confirmar se o usuário informou o número de argumentos corretos, bem como, a captura das informações transmitidas pela linha de comando.

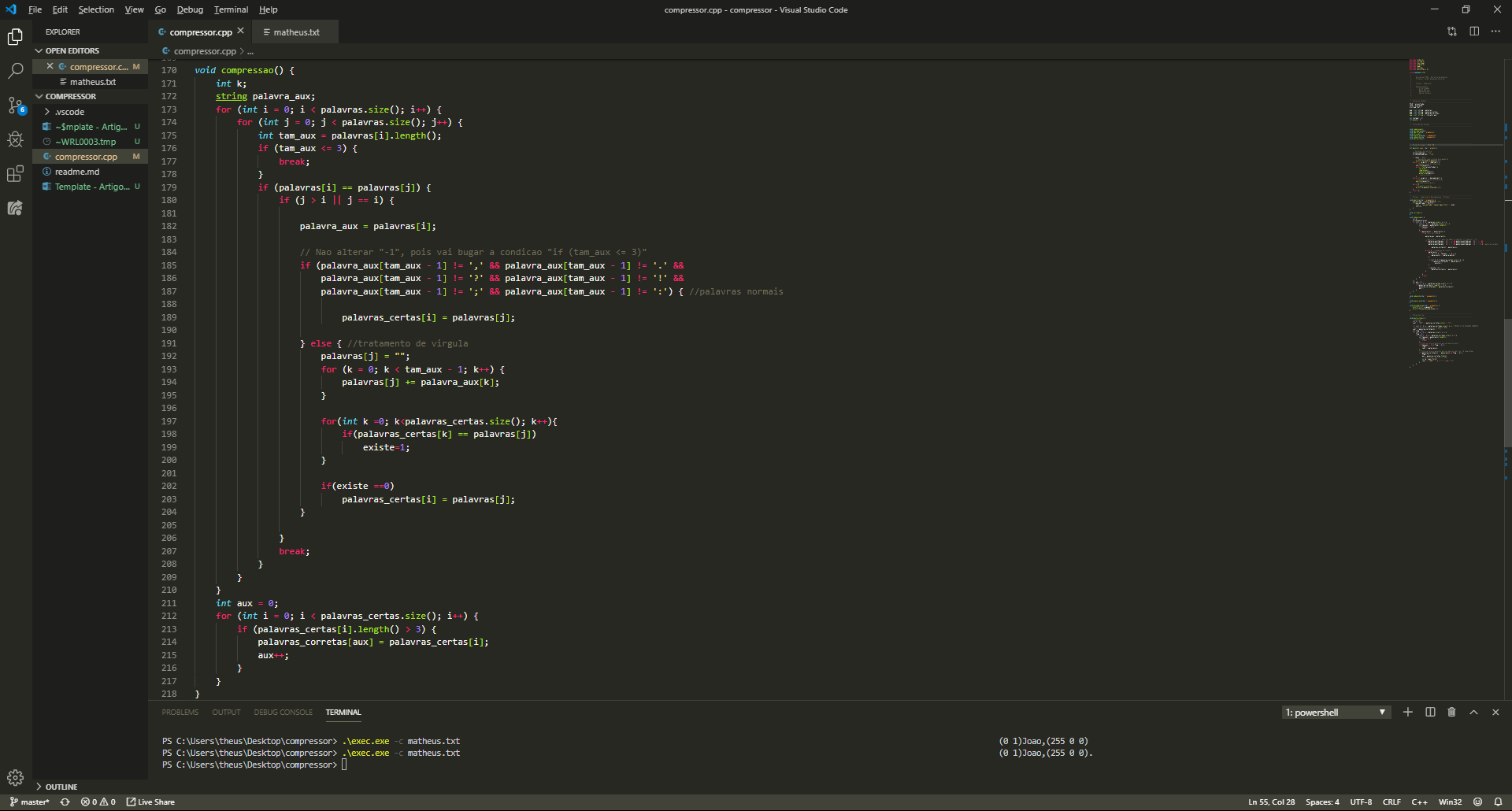


***Figura 3*** *– Função MAIN*

A Figura 3 demonstra todo o corpo do programa, desta forma, torna-se possível explicar como cada uma das funções que realizam a compactação (C1) funcionam.

A primeira função a ser chamada durante a compactação do arquivo é a mais simples dentre todas, é a função “***abertura(argv[2])***” que realiza a abertura do arquivo de entrada, ou seja, ela lê o “***nome\_arquivo.txt***” que o usuário informou via linha de comando.

Após a abertura do arquivo, entra-se em uma estrutura de repetição que tem como condição de parada o final do arquivo. Então dentro dessa estrutura de repetição (*while*) há chamada para quatro função, que são: “***ler\_arq()***”, “***compressao()***”, “***cabeçalho(argv[2])***” e “***gravar\_arq(argv[2])***”. Como os nomes são autoexplicativos, pode-se inferir que, ao entrar na estrutura de repetição, haverá a leitura de 4096 *Bytes,* então o texto lido será compactado pela função “***compressao()***” e, por fim, as duas últimas funções serão responsáveis por gravar o texto compactado em um outro arquivo, com o mesmo nome do primeiro, porém, com a extensão “***.cmp***”.

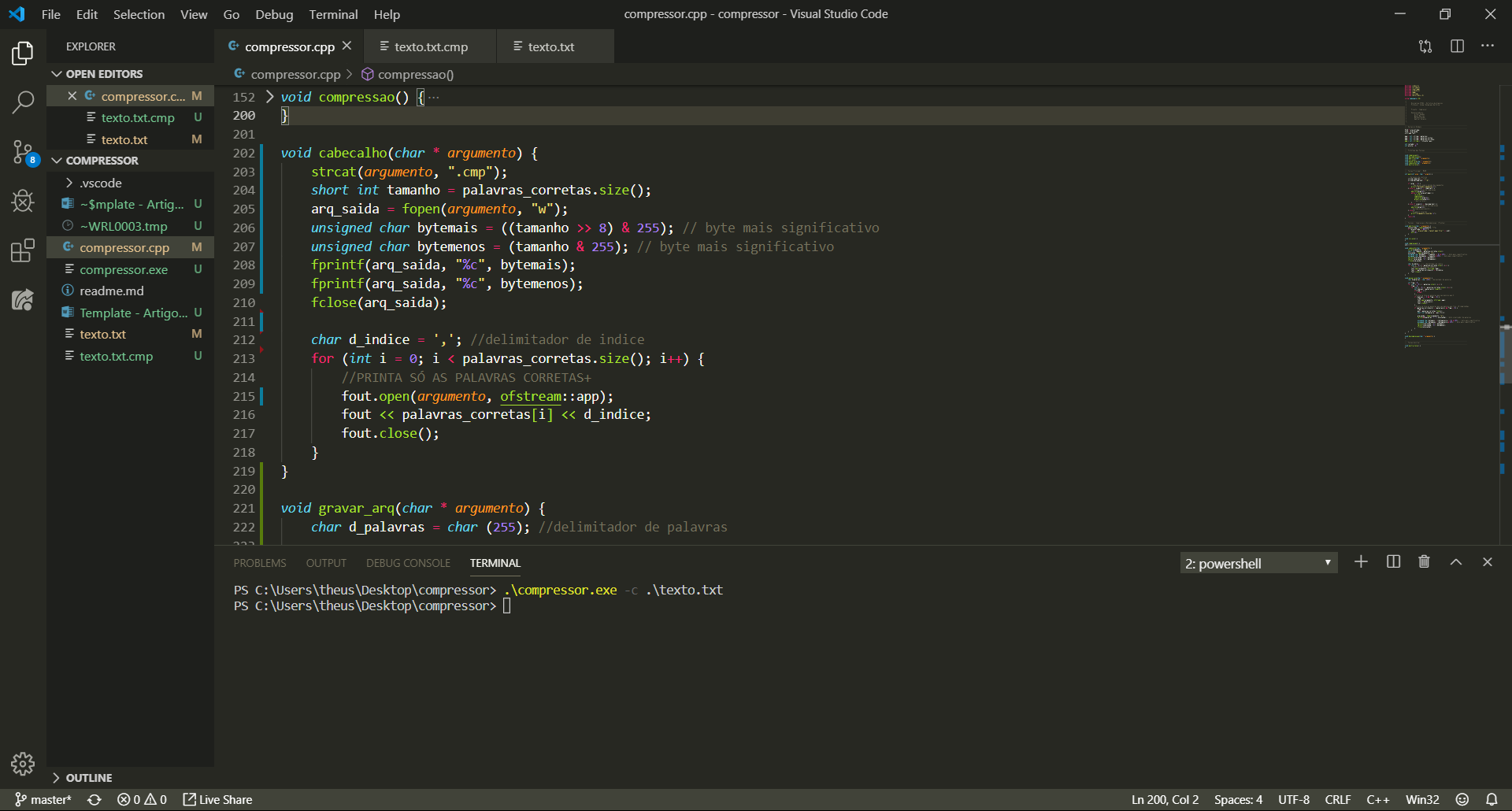


***Figura 4*** *– Função de compressão*

A Figura 4 retrata a função de compressão, é ela responsável por comprimir palavra por palavra do *map* que guardou cada uma das palavras do texto original. Assim sendo, para realizar a compressão, há duas estruturas de repetição que percorrerão todas as posições do *map* (são necessárias dua estruturas de repetição para comparar cada uma das palavras com todas as outras), então, a primeira condição, “***if(tam\_aux <= 3)***”, existe para verificar se a primeira palavra possui mais de três caracteres. Caso a condição seja verdadeira o laço é interrompido, caso a condição seja falsa parte-se para a próxima condição.

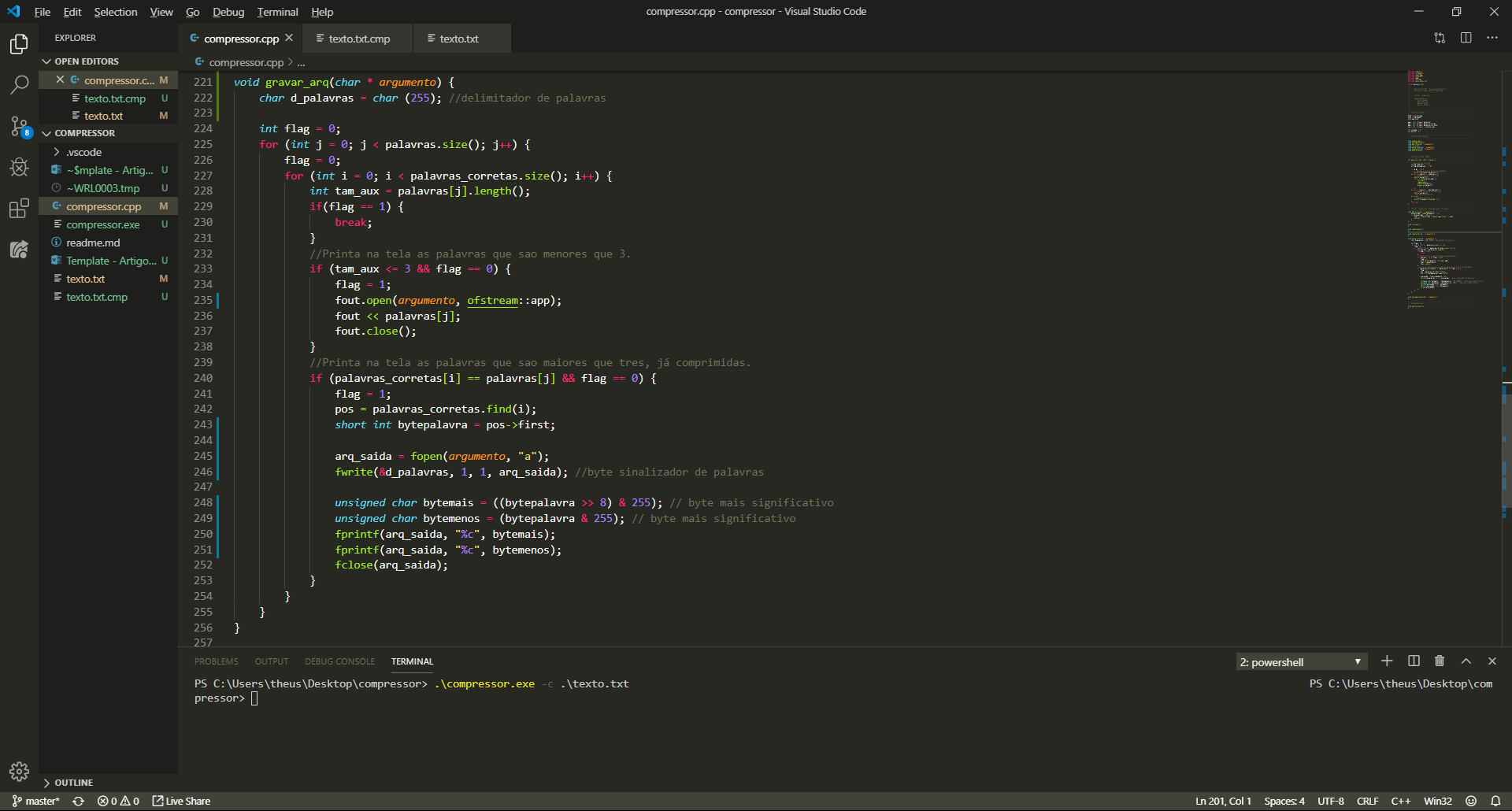
Esta condição (“***if(palavras[i] == palavras[j])***”), por sua vez, faz a comparação entre cada uma das palavras do *map* com todas as outras, então se a condição for verdadeira e as palavras não estiverem na mesma posição (“***if(j > i || j == i)***”), a palavra em questão é atribuída a uma variável auxiliar para que seja possível iniciar o tratamento de pontuações. Caso o último caractere da variável auxiliar não seja um caractere de pontuação a palavra do primeiro *map* (palavras) é atribuída a um novo *map* (palavras\_certas) que possui as palavras sem nenhuma pontuação. Caso contrário, isto é, se o ultimo caractere da palavra auxiliar for um caractere especial inicia-se o tratamento das palavras que possuem pontuação. Esse tratamento consiste em retirar a pontuação da ultimo caractere e atribuir a palavra do primeiro *map* (palavras) para o segundo *map* (palavras\_certas).

Por fim, ao terminar a rotina das duas condições anteriores, entra-se em um novo laço de repetição. Este laço existe para que todos os valores que estão no segundo map (palavras\_certas) sejam atribuídos ao terceiro *map* (palavras\_corretas), isso é totalmente necessário, pois o segundo *map* (palavras\_certas) estava comparando posições que não existiam, então, ao comparar uma posição que não existe em um *map* ela é criada automaticamente.



***Figura 5*** *– Função de cabeçalho*

A Figura 5 retrata função “***cabeçalho(char \* argumento)***”, primeiramente esta função concatena duas *strings*, isto é, o nome do arquivo de texto original com a extensão padrão “.cmp”, para que seja possível criar um arquivo de saída padronizado. Logo após, esta função simplesmente pega o tamanho (**short int tamanho**) do terceiro *map* (palavras\_corretas), transforma esse tamanho em dois bytes, sendo o primeiro o mais significativo e o segundo o menos significativo e os imprime em um arquivo. Então, a última coisa a fazer é printar todas o terceiro map (palavras\_corretas) no arquivo, separando-os com uma vírgula.



***Figura 6*** *– Função de gravar em arquivo*

A Figura 6 retrata função “***gravar\_arq(char \* argumento)***”, esta é a função responsável por gravar no arquivo as palavras compactadas ou não em suas respectivas posições. Para isso, primeiramente, assim como na função de compressão (Figura 4) há duas estruturas de repetição e, aninhadas a elas estão duas principais condições, que são: “**if(tam\_aux <= 3 && flag == 0)**” e “**if(palavras\_corretas[i] == palavras\_corretas[j] && flag == 0)**”.

Caso a primeira condição seja verdadeira é porque nesse laço da iteração, a palavra em questão possui três caracteres ou menos, logo, ela deve ser imprimida no arquivo em seu estado natural, isto é, sem nenhum tipo de compactação (lembrando que em nosso programa os caracteres especiais e espaços são estão contidos no primeiro *map* [palavras]). Caso a condição seja falsa parte-se para segunda condição que verifica em que posição as palavras do terceiro *map* (palavras\_corretas) estão dentro do primeiro *map* (palavras), então, ao achar a posição, basta imprimir no arquivo o delimitador de palavrasm isto é, o *byte* 255 e a chave do terceiro *map* (palavras\_corretas), lembrando que antes de imprimir a chave do map no arquivo é preciso transformar a chave em dois bytes, sendo o primeiro o mais significativo e o segundo o menos significativo.

**RESULTADOS**

Bla bla bla bla...

**DISCUSSÃO E CONCLUSÃO**

Bla bla bla bla...