Contents

1 Basic Test Results

```
Running...
Opening tar file
ChangeBase.c
CheckParenthesis.c
OK
Tar extracted O.K.
Checking files...
OK
Making sure files are not empty...
OK
Compilation check...
Compiling...
OK
Compiling...
OK
Compilation seems OK! Check if you got warnings!

========================
= Checking coding style =
========================
 ** Total Violated Rules      : 0
 ** Total Errors Occurs       : 0
 ** Total Violated Files Count: 0
```

2 ChangeBase.c

```c
/**
 * @file ChangeBase.c
 * @author Itai Tagar <itagar>
 * @version 2.0
 * @date 09 Aug 2016
 *
 * @brief A program that convert a given number from one base representation to another.
 *
 * @section LICENSE
 * This program is free to use in every operation system.
 *
 * @section DESCRIPTION
 * A program that convert a given number from one base representation to another.
 * Input:       One input that holds the given number, it's current base representation and the
 *              new base we want to convert to. The input comes from the user in the format of -
 *              <original base>^<new base>^<the number in original base>^
 * Process:     The program analyze if the input is valid, an invalid state is where the given
 *              number cannot be represented with the given original base.
 *              After validating the input, the program convert the number to the new base
 *              representation and prints it out to the screen.
 * Output:      The converted number is printed to the screen if the input was valid.
 *              An error message in case of bad input.
 */


/*-----=  Includes  =-----*/


#include <stdio.h>


/*-----=  Definitions  =-----*/


/**
 * @def VALID_STATE 0
 * @brief A Flag for valid state during the program run.
 */
#define VALID_STATE 0

/**
 * @def INVALID_STATE 1
 * @brief A Flag for invalid state during the program run.
 */
#define INVALID_STATE 1

/**
 * @def STANDARD_BASE 10
 * @brief A Macro that sets the standard base which we usually use.
 */
#define STANDARD_BASE 10

/**
 * @def TRUE 1
 * @brief A Flag for true statement.
 */
#define TRUE 1

/**
```

```c
 * @def FALSE 0
 * @brief A Flag for false statement.
 */
#define FALSE 0

/**
 * @def MAX_RESULT_SIZE 20
 * @brief A Macro that sets the maximum number of digits for the result number after conversion.
 */
#define MAX_RESULT_SIZE 20

/**
 * @def INVALID_INPUT_MESSAGE "invalid!!\n"
 * @brief A Macro that sets the output message for invalid user input.
 */
#define INVALID_INPUT_MESSAGE "invalid!!\n"


/*----=  Forward Declarations  =-----*/


/**
 * @brief Performs the base conversion from a given bases for the desired given number to convert.
 *        The function first convert the number to be represented in base 10, and then convert
 *        from base 10 to the desired new base.
 *        Explanation of the Algorithm: In the description of this function's definition.
 * @param originalBase The base in which the given number is currently represented.
 * @param newBase The base to convert the given number representation to.
 * @param number The given number, represented in the original base, that should be converted.
 * @param result The path to store the conversion result in.
 * @return char array holding the converted number.
 */
char * baseConverter(int const originalBase, int const newBase, int number, char * result);

/**
 * @brief An Helper function for the Base Converter function.
 *        This function perform the actual base conversion from the given number to decimal,
 *        i.e. represented in base 10.
 * @param originalBase The base in which the given number is currently represented.
 * @param number The given number, represented in the original base, that should be converted.
 * @return The number represented in base 10 as decimal.
 */
int decimalConverter(int const originalBase, int number);

/**
 * @brief An Helper function for the Base Converter function.
 *        This function perform the actual base conversion, assuming the original base is 10,
 *        and convert the given number to the new base.
 *        The function updates the given result array with the converted number.
 * @param newBase The base to convert the given number representation to.
 * @param number The given number, represented in the original base, that should be converted.
 * @param result The path to store the conversion result in.
 */
void baseConverterHelper(int const newBase, int number, char * result);

/**
 * @brief A Power operator. Raises the base in the power of degree.
 * @param base The base of the power.
 * @param degree The degree of the power.
 * @return The result of the base raised to the degree.
 */
int power(int const base, int const degree);

/**
 * @brief Verify that the given number in the user input can be represented in the
 *        given original base.
 * @param originalBase The given original base in the user input.
 * @param number The given number in the user input.
 * @return 0 if the input is invalid, 1 otherwise.
 */
int checkInput(int const originalBase, int number);

/**
 * @brief Prints the given conversion result to the standard output.
 *        During the conversion, the result is stored backwards, so this function determines which
 *        index is the last index in the result array that contains data, and from this points it
 *        prints the data all the way back.
 * @param The result of the converted number.
 */
void printResult(char * result);


/*----=  Main  =-----*/


/**
 * @brief The main function that runs the program. The function receive input from the user
 *        and perform the base conversion using the base conversion functions.
 *        The function determines if the input is valid, and if so it prints the result of the
 *        base conversion. If the input is invalid, the function will print an error message.
 * @return 0 when the program ran successfully, 1 otherwise.
 */
int main()
{
    // Initialize variables.
    int originalBase = 0;
    int newBase = 0;
    int number = 0;
    char result[MAX_RESULT_SIZE + 1] = {};

    // Receive input from user parse it to the relevant variables.
    scanf("%d^%d^%d^", &originalBase, &newBase, &number);

    // If the given number is 0, it does not matter what are the bases, the result will be 0.
    if (number == 0)
    {
        printf("%d\n", number);
    }
    else
    {
        // Verify input, Convert the number and print the result.
        if (checkInput(originalBase, number))
        {
            printResult(baseConverter(originalBase, newBase, number, result));
        }
        else
        {
            fprintf(stderr, INVALID_INPUT_MESSAGE);
            return INVALID_STATE;
        }
    }

    return VALID_STATE;
}


/*----=  Base Conversion  =-----*/


/**
 * @brief Performs the base conversion from a given bases for the desired given number to convert.
 *        The function first convert the number to be represented in base 10, and then convert
 *        from base 10 to the desired new base.
 *        Explanation of the Algorithm:
 *        The algorithm used in order to convert the number is as follow:
 *        We take each digit in the given number, starting from the lowest one, and
 *        perform Euclidean Division of this digit with the new base we want to convert to.
 *        Then, we take the remainder and multiply it with the original base raised to the power
 *        of the digit index.
 *        We take the quotient from the Euclidean Division and perform the same actions on this
 *        number. We keep doing so until the quotient is equals to 0.
 *        We sum up all of our calculations of the remainders multiplied by the original base
 *        powers, and this is the result for the converted number.
 *        The running time complexity of this algorithm is O(n) where n is the number of digits
 *        in the given number to convert.
 *        In the main call for this algorithm, we perform a conversion to base 10 first, and then
 *        convert from base 10 to the new base. So we call this function twice, so the
 *        running time complexity is O(n) + O(n), i.e. O(n).
 *        This algorithm was taught during Linear Algebra class, in order to represent one
 *        polynomial another polynomial's degrees.
 * @param originalBase The base in which the given number is currently represented.
 * @param newBase The base to convert the given number representation to.
 * @param number The given number, represented in the original base, that should be converted.
 * @param result The path to store the conversion result in.
 * @return char array holding the converted number.
 */
char * baseConverter(int const originalBase, int const newBase, int number, char * result)
{
    int baseTenNumber = decimalConverter(originalBase, number);  // Convert to Decimal.
    baseConverterHelper(newBase, baseTenNumber, result);
    return result;
}

/**
 * @brief An Helper function for the Base Converter function.
 *        This function perform the actual base conversion from the given number to decimal,
 *        i.e. represented in base 10.
 * @param originalBase The base in which the given number is currently represented.
 * @param number The given number, represented in the original base, that should be converted.
 * @return The number represented in base 10 as decimal.
 */
int decimalConverter(int const originalBase, int number)
{
    int result = 0;

    int index = 0;
    while (number != 0)
    {
        int currentDigit = number % STANDARD_BASE;
        result += (currentDigit * (power(originalBase, index)));
        number /= STANDARD_BASE;
        index++;
    }

    return result;
}

/**
 * @brief An Helper function for the Base Converter function.
 *        This function perform the actual base conversion, assuming the original base is 10,
 *        and convert the given number to the new base.
 *        The function updates the given result array with the converted number.
 * @param newBase The base to convert the given number representation to.
 * @param number The given number, represented in the original base, that should be converted.
 * @param result The path to store the conversion result in.
 */
void baseConverterHelper(int const newBase, int number, char * result)
{
    int index = 0;
    while (number != 0)
    {
        int currentDigit = number % newBase;
        result[index] = (char)(currentDigit + '0');
        number /= newBase;
```

```c
        index++;
    }
}

/**
 * @brief A Power operator. Raises the base in the power of degree.
 * @param base The base of the power.
 * @param degree The degree of the power.
 * @return The result of the base raised to the degree.
 */
int power(int const base, int const degree)
{
    if (degree == 0)
    {
        return 1;
    }
    else
    {
        return (power(base, degree - 1)) * base;
    }
}


/*----=  Input Handling  =-----*/


/**
 * @brief Verify that the given number in the user input can be represented in the
 *        given original base.
 * @param originalBase The given original base in the user input.
 * @param number The given number in the user input.
 * @return 0 if the input is invalid, 1 otherwise.
 */
int checkInput(int const originalBase, int number)
{
    while (number != 0)
    {
        int currentDigit = number % STANDARD_BASE;
        if (currentDigit >= originalBase)
        {
            return FALSE;
        }
        number /= STANDARD_BASE;
    }
    return TRUE;
}


/*----=  Output Handling  =-----*/


/**
 * @brief Prints the given conversion result to the standard output.
 *        During the conversion, the result is stored backwards, so this function determines which
 *        index is the last index in the result array that contains data, and from this points it
 *        prints the data all the way back.
 * @param The result of the converted number.
 */
void printResult(char * result)
{
    // Determine the indices of result that contain data.
    int i = 0;
    while (result[i] != 0)
    {
        i++;
    }

    // Prints the converted number in the required order.
    i--;  // index 'i' is currently at the '\0' character, we need to take 1 step backwards.
    for ( ; i >= 0; --i)
    {
        printf("%c", result[i]);
    }
    printf("\n");
}
```

3 CheckParenthesis.c

```c
/**
 * @file CheckParenthesis.c
 * @author Itai Tagar <itagar>
 * @version 1.2
 * @date 09 Aug 2016
 *
 * @brief A program that verify text files that satisfies a desired parenthesis structure.
 *
 * @section LICENSE
 * This program is free to use in every operation system.
 *
 * @section DESCRIPTION
 * A program that verify text files that satisfies a desired parenthesis structure.
 * Input:       A name or a path to a text file.
 * Process:     Validates input, if the input is valid the program starts to analyze the text file
 *              for determine if the structure of parenthesis is valid or invalid.
 *              If the file is invalid the program ends with an error message.
 * Output:      A message that states the file analysis results, if the input was valid.
 *              An error message in case of bad input.
 */


/*-----=  Includes  =-----*/


#include <stdio.h>


/*-----=  Definitions  =-----*/


/**
 * @def VALID_STATE 0
 * @brief A Flag for valid state during the program run.
 */
#define VALID_STATE 0

/**
 * @def INVALID_STATE 1
 * @brief A Flag for invalid state during the program run.
 */
#define INVALID_STATE 1

/**
 * @def VALID_ARGUMENTS_NUMBER 2
 * @brief A Macro that sets the valid number of arguments for this program.
 */
#define VALID_ARGUMENTS_NUMBER 2

/**
 * @def INVALID_ARGUMENTS_MESSAGE "Please supply a file!\nusage: CheckParenthesis <filename>\n"
 * @brief A Macro that sets the output message for invalid arguments.
 */
#define INVALID_ARGUMENTS_MESSAGE "Please supply a file!\nusage: CheckParenthesis <filename>\n"

/**
 * @def FILE_NAME_INDEX 1
 * @brief A Macro that sets the index of the File name in the arguments array.
 */
```

```c
#define FILE_NAME_INDEX 1

/**
 * @def INVALID_FILE_ARGUMENTS_MESSAGE "Error! trying to open the file %s\n"
 * @brief A Macro that sets the output message for an invalid File argument.
 */
#define INVALID_FILE_ARGUMENTS_MESSAGE "Error! trying to open the file %s\n"

/**
 * @def VALID_FILE "ok\n"
 * @brief A Macro that sets the output message for a valid File.
 */
#define VALID_FILE "ok\n"

/**
 * @def INVALID_FILE "bad structure\n"
 * @brief A Macro that sets the output message for a invalid File.
 */
#define INVALID_FILE "bad structure\n"

/**
 * @def INITIAL_SCOPE_NUMBER 0
 * @brief A Macro that sets the initial scope number in a given File.
 */
#define INITIAL_SCOPE_NUMBER 0

/**
 * @def OPEN_ROUND '('
 * @brief A Flag for the Round Opening-Parenthesis character.
 */
#define OPEN_ROUND '('

/**
 * @def CLOSE_ROUND ')'
 * @brief A Flag for the Round Closing-Parenthesis character.
 */
#define CLOSE_ROUND ')'

/**
 * @def OPEN_SQUARE '['
 * @brief A Flag for the Square Opening-Parenthesis character.
 */
#define OPEN_SQUARE '['

/**
 * @def CLOSE_SQUARE ']'
 * @brief A Flag for the Square Closing-Parenthesis character.
 */
#define CLOSE_SQUARE ']'

/**
 * @def OPEN_TRIANGLE '<'
 * @brief A Flag for the Triangle Opening-Parenthesis character.
 */
#define OPEN_TRIANGLE '<'

/**
 * @def CLOSE_TRIANGLE '>'
 * @brief A Flag for the Triangle Closing-Parenthesis character.
 */
#define CLOSE_TRIANGLE '>'

/**
 * @def OPEN_CURLY '{'
 * @brief A Flag for the Curly Opening-Parenthesis character.
 */
#define OPEN_CURLY '{'

```

```c
/**
 * @def CLOSE_CURLY '}'
 * @brief A Flag for the Curly Closing-Parenthesis character.
 */
#define CLOSE_CURLY '}'


/*----=  Forward Declarations  =-----*/


/**
 * @brief Analyze the results of the 'checkFile' functions, and perform the
 *        required actions for each scenario.
 * @param checkFileResult The given result of the 'checkFile' functions.
 */
void analyzeResults(int const checkFileResult);

/**
 * @brief Checks the given File for valid parenthesis structure.
 * @param pFile The given File to check.
 * @return 0 if the given File satisfies the required parenthesis structure, 1 otherwise.
 */
int checkFile(FILE * const pFile);

/**
 * @brief Checks the given File for valid parenthesis structure.
 *        This function perform recursive calls each time a new parenthesis is opened.
 * @param currentType The current type of Opening-Parenthesis in this call of the function.
 * @param pFile The given File to check.
 * @return 0 if the given File satisfies the required parenthesis structure, 1 otherwise.
 */
int checkFileHelper(char const currentType, FILE * const pFile);

/**
 * @brief Checks if a given 2 parenthesis are matching each other (i.e. one closes the other).
 * @param close The Closing-Parenthesis character.
 * @param open The Opening-Parenthesis character.
 * @return 0 if the given 2 parenthesis are matching each other, 1 otherwise.
 */
int checkMatchingParenthesis(char const close, char const open);


/*----=  Main  =-----*/


/**
 * @brief The main function that runs the program.
 *        It receives arguments from the user and if the arguments are valid, it runs the File
 *        Analysis.
 * @param argc The number of given arguments.
 * @param argv[] The arguments from the user.
 * @return 0 if the given File is a text file which satisfies the required
 *         parenthesis structure, 1 otherwise.
 */
int main(int argc, char * argv[])
{

    // Check valid arguments.
    if (argc != VALID_ARGUMENTS_NUMBER)
    {
        fprintf(stderr, INVALID_ARGUMENTS_MESSAGE);
        return INVALID_STATE;
    }
    else
    {
        // Receive the File to check.
        FILE * pFile;
        pFile = fopen(argv[FILE_NAME_INDEX], "r");

        // In case of a bad File.
        if (pFile == 0)
        {
            fprintf(stderr, INVALID_FILE_ARGUMENTS_MESSAGE, argv[FILE_NAME_INDEX]);
            fclose(pFile);
            return INVALID_STATE;
        }

        // Analyze the File and close its Stream.
        int checkFileResult = checkFile(pFile);
        fclose(pFile);

        // Analyze the results.
        analyzeResults(checkFileResult);
        return VALID_STATE;
    }
}


/*-----=  Analyze File  =-----*/


/**
 * @brief Analyze the results of the 'checkFile' functions, and perform the
 *        required actions for each scenario.
 * @param checkFileResult The given result of the 'checkFile' functions.
 */
void analyzeResults(int const checkFileResult)
{
    if (!(checkFileResult))  // If the File is valid, 'checkFileResult' will be equal to 0.
    {
        printf(VALID_FILE);
    }
    else
    {
        printf(INVALID_FILE);
    }
}

/**
 * @brief Checks the given File for valid parenthesis structure.
 * @param pFile The given File to check.
 * @return 0 if the given File satisfies the required parenthesis structure, 1 otherwise.
 */
int checkFile(FILE * const pFile)
{
    return checkFileHelper(EOF, pFile);
}

/**
 * @brief Checks the given File for valid parenthesis structure.
 *        This function perform recursive calls each time a new parenthesis is opened.
 * @param currentType The current type of Opening-Parenthesis in this call of the function.
 * @param pFile The given File to check.
 * @return 0 if the given File satisfies the required parenthesis structure, 1 otherwise.
 */
int checkFileHelper(char const currentType, FILE * const pFile)
{
    static int scopeCounter = INITIAL_SCOPE_NUMBER;

    int currentChar;  // The current character in the given File.

    while ((currentChar = fgetc(pFile)) != EOF)
    {
        // In case we reached any kind of Opening-Parenthesis, we enter a recursive call
        // and increase the 'scopeCounter' by 1.
        if (currentChar == OPEN_ROUND)
        {
            scopeCounter++;
            checkFileHelper(OPEN_ROUND, pFile);
        }
        else if (currentChar == OPEN_SQUARE)
        {
            scopeCounter++;
            checkFileHelper(OPEN_SQUARE, pFile);
        }
        else if (currentChar == OPEN_TRIANGLE)
        {
            scopeCounter++;
            checkFileHelper(OPEN_TRIANGLE, pFile);
        }
        else if (currentChar == OPEN_CURLY)
        {
            scopeCounter++;
            checkFileHelper(OPEN_CURLY, pFile);
        }

        // In case we reached any kind of Closing-Parenthesis, we determine if it is valid.
        // If it is valid we exit the current recursive call and decrease the 'scopeCounter' by 1.
        // If it is invalid, we exit the recursive call with the value 1.
        if (currentChar == CLOSE_ROUND || currentChar == CLOSE_SQUARE ||
            currentChar == CLOSE_TRIANGLE || currentChar == CLOSE_CURLY)
        {
            if (!(checkMatchingParenthesis((char) currentChar, currentType)))
            {
                scopeCounter--;
                return VALID_STATE;
            }
            else
            {
                return INVALID_STATE;
            }
        }
    }

    // In case we reached the end of the File, we check that there are no Opening-Parenthesis
    // left unclosed, using the 'scopeCounter'.
    if (scopeCounter == INITIAL_SCOPE_NUMBER)
    {
        return VALID_STATE;
    }
    else
    {
        return INVALID_STATE;
    }
}

/**
 * @brief Checks if a given 2 parenthesis are matching each other (i.e. one closes the other).
 * @param close The Closing-Parenthesis character.
 * @param open The Opening-Parenthesis character.
 * @return 0 if the given 2 parenthesis are matching each other, 1 otherwise.
 */
int checkMatchingParenthesis(char const close, char const open)
{
    switch (close)
    {
        case (CLOSE_ROUND):
            if (open != OPEN_ROUND)
            {
                return INVALID_STATE;
            }
            else
            {
                return VALID_STATE;
            }

        case (CLOSE_SQUARE):
            if (open != OPEN_SQUARE)
            {
                return INVALID_STATE;
            }
            else
            {
                return VALID_STATE;
            }

        case (CLOSE_TRIANGLE):
            if (open != OPEN_TRIANGLE)
            {
                return INVALID_STATE;
            }
            else
            {
                return VALID_STATE;
            }

        case (CLOSE_CURLY):
            if (open != OPEN_CURLY)
            {
                return INVALID_STATE;
            }
            else
            {
                return VALID_STATE;
            }

        default:
            return INVALID_STATE;
    }
}
```