

# Contents

1	Basic Test Results	2
2	ChangeBase.c	3
3	CheckParenthesis.c	9

# 1 Basic Test Results

```
1  Running...
2  Opening tar file
3  ChangeBase.c
4  CheckParenthesis.c
5  OK
6  Tar extracted O.K.
7  Checking files...
8  OK
9  Making sure files are not empty...
10 OK
11 Compilation check...
12 Compiling...
13 OK
14 Compiling...
15 OK
16 Compilation seems OK! Check if you got warnings!
17
18 =====
19 = Checking coding style =
20 =====
21 ** Total Violated Rules      : 0
22 ** Total Errors Occurs      : 0
23 ** Total Violated Files Count: 0
```

## 2 ChangeBase.c

```
1  /**
2   * @file ChangeBase.c
3   * @author Itai Tagar <itagar>
4   * @version 2.0
5   * @date 09 Aug 2016
6   *
7   * @brief A program that convert a given number from one base representation to another.
8   *
9   * @section LICENSE
10  * This program is free to use in every operation system.
11  *
12  * @section DESCRIPTION
13  * A program that convert a given number from one base representation to another.
14  * Input:      One input that holds the given number, it's current base representation and the
15  *             new base we want to convert to. The input comes from the user in the format of -
16  *             <original base>^<new base>^<the number in original base>^
17  * Process:    The program analyze if the input is valid, an invalid state is where the given
18  *             number cannot be represented with the given original base.
19  *             After validating the input, the program convert the number to the new base
20  *             representation and prints it out to the screen.
21  * Output:    The converted number is printed to the screen if the input was valid.
22  *            An error message in case of bad input.
23  */
24
25
26  /*----- Includes -----*/
27
28
29  #include <stdio.h>
30
31
32  /*----- Definitions -----*/
33
34
35  /**
36   * @def VALID_STATE 0
37   * @brief A Flag for valid state during the program run.
38   */
39  #define VALID_STATE 0
40
41  /**
42   * @def INVALID_STATE 1
43   * @brief A Flag for invalid state during the program run.
44   */
45  #define INVALID_STATE 1
46
47  /**
48   * @def STANDARD_BASE 10
49   * @brief A Macro that sets the standard base which we usually use.
50   */
51  #define STANDARD_BASE 10
52
53  /**
54   * @def TRUE 1
55   * @brief A Flag for true statement.
56   */
57  #define TRUE 1
58
59  /**
```

```

60  * @def FALSE 0
61  * @brief A Flag for false statement.
62  */
63  #define FALSE 0
64
65  /**
66   * @def MAX_RESULT_SIZE 20
67   * @brief A Macro that sets the maximum number of digits for the result number after conversion.
68   */
69  #define MAX_RESULT_SIZE 20
70
71  /**
72   * @def INVALID_INPUT_MESSAGE "invalid!!\n"
73   * @brief A Macro that sets the output message for invalid user input.
74   */
75  #define INVALID_INPUT_MESSAGE "invalid!!\n"
76
77
78  /*----- Forward Declarations -----*/
79
80
81  /**
82   * @brief Performs the base conversion from a given bases for the desired given number to convert.
83   *      The function first convert the number to be represented in base 10, and then convert
84   *      from base 10 to the desired new base.
85   *      Explanation of the Algorithm: In the description of this function's definition.
86   * @param originalBase The base in which the given number is currently represented.
87   * @param newBase The base to convert the given number representation to.
88   * @param number The given number, represented in the original base, that should be converted.
89   * @param result The path to store the conversion result in.
90   * @return char array holding the converted number.
91   */
92  char * baseConverter(int const originalBase, int const newBase, int number, char * result);
93
94  /**
95   * @brief An Helper function for the Base Converter function.
96   *      This function perform the actual base conversion from the given number to decimal,
97   *      i.e. represented in base 10.
98   * @param originalBase The base in which the given number is currently represented.
99   * @param number The given number, represented in the original base, that should be converted.
100  * @return The number represented in base 10 as decimal.
101  */
102  int decimalConverter(int const originalBase, int number);
103
104  /**
105   * @brief An Helper function for the Base Converter function.
106   *      This function perform the actual base conversion, assuming the original base is 10,
107   *      and convert the given number to the new base.
108   *      The function updates the given result array with the converted number.
109   * @param newBase The base to convert the given number representation to.
110   * @param number The given number, represented in the original base, that should be converted.
111   * @param result The path to store the conversion result in.
112   */
113  void baseConverterHelper(int const newBase, int number, char * result);
114
115  /**
116   * @brief A Power operator. Raises the base in the power of degree.
117   * @param base The base of the power.
118   * @param degree The degree of the power.
119   * @return The result of the base raised to the degree.
120   */
121  int power(int const base, int const degree);
122
123  /**
124   * @brief Verify that the given number in the user input can be represented in the
125   *      given original base.
126   * @param originalBase The given original base in the user input.
127   * @param number The given number in the user input.

```

```

128  * @return 0 if the input is invalid, 1 otherwise.
129  */
130  int checkInput(int const originalBase, int number);
131
132  /**
133   * @brief Prints the given conversion result to the standard output.
134   *      During the conversion, the result is stored backwards, so this function determines which
135   *      index is the last index in the result array that contains data, and from this points it
136   *      prints the data all the way back.
137   * @param The result of the converted number.
138   */
139  void printResult(char * result);
140
141
142  /*----- Main -----*/
143
144
145  /**
146   * @brief The main function that runs the program. The function receive input from the user
147   *      and perform the base conversion using the base conversion functions.
148   *      The function determines if the input is valid, and if so it prints the result of the
149   *      base conversion. If the input is invalid, the function will print an error message.
150   * @return 0 when the program ran successfully, 1 otherwise.
151   */
152  int main()
153  {
154      // Initialize variables.
155      int originalBase = 0;
156      int newBase = 0;
157      int number = 0;
158      char result[MAX_RESULT_SIZE + 1] = {};
159
160      // Receive input from user parse it to the relevant variables.
161      scanf("%d~%d~%d", &originalBase, &newBase, &number);
162
163      // If the given number is 0, it does not matter what are the bases, the result will be 0.
164      if (number == 0)
165      {
166          printf("%d\n", number);
167      }
168      else
169      {
170          // Verify input, Convert the number and print the result.
171          if (checkInput(originalBase, number))
172          {
173              printResult(baseConverter(originalBase, newBase, number, result));
174          }
175          else
176          {
177              fprintf(stderr, INVALID_INPUT_MESSAGE);
178              return INVALID_STATE;
179          }
180      }
181
182      return VALID_STATE;
183  }
184
185
186  /*----- Base Conversion -----*/
187
188
189  /**
190   * @brief Performs the base conversion from a given bases for the desired given number to convert.
191   *      The function first convert the number to be represented in base 10, and then convert
192   *      from base 10 to the desired new base.
193   *      Explanation of the Algorithm:
194   *      The algorithm used in order to convert the number is as follow:
195   *      We take each digit in the given number, starting from the lowest one, and

```

```

196 *      perform Euclidean Division of this digit with the new base we want to convert to.
197 *      Then, we take the remainder and multiply it with the original base raised to the power
198 *      of the digit index.
199 *      We take the quotient from the Euclidean Division and perform the same actions on this
200 *      number. We keep doing so until the quotient is equals to 0.
201 *      We sum up all of our calculations of the remainders multiplied by the original base
202 *      powers, and this is the result for the converted number.
203 *      The running time complexity of this algorithm is  $O(n)$  where  $n$  is the number of digits
204 *      in the given number to convert.
205 *      In the main call for this algorithm, we perform a conversion to base 10 first, and then
206 *      convert from base 10 to the new base. So we call this function twice, so the
207 *      running time complexity is  $O(n) + O(n)$ , i.e.  $O(n)$ .
208 *      This algorithm was taught during Linear Algebra class, in order to represent one
209 *      polynomial another polynomial's degrees.
210 * @param originalBase The base in which the given number is currently represented.
211 * @param newBase The base to convert the given number representation to.
212 * @param number The given number, represented in the original base, that should be converted.
213 * @param result The path to store the conversion result in.
214 * @return char array holding the converted number.
215 */
216 char * baseConverter(int const originalBase, int const newBase, int number, char * result)
217 {
218     int baseTenNumber = decimalConverter(originalBase, number); // Convert to Decimal.
219     baseConverterHelper(newBase, baseTenNumber, result);
220     return result;
221 }
222
223 /**
224 * @brief An Helper function for the Base Converter function.
225 *      This function perform the actual base conversion from the given number to decimal,
226 *      i.e. represented in base 10.
227 * @param originalBase The base in which the given number is currently represented.
228 * @param number The given number, represented in the original base, that should be converted.
229 * @return The number represented in base 10 as decimal.
230 */
231 int decimalConverter(int const originalBase, int number)
232 {
233     int result = 0;
234
235     int index = 0;
236     while (number != 0)
237     {
238         int currentDigit = number % STANDARD_BASE;
239         result += (currentDigit * (power(originalBase, index)));
240         number /= STANDARD_BASE;
241         index++;
242     }
243
244     return result;
245 }
246
247 /**
248 * @brief An Helper function for the Base Converter function.
249 *      This function perform the actual base conversion, assuming the original base is 10,
250 *      and convert the given number to the new base.
251 *      The function updates the given result array with the converted number.
252 * @param newBase The base to convert the given number representation to.
253 * @param number The given number, represented in the original base, that should be converted.
254 * @param result The path to store the conversion result in.
255 */
256 void baseConverterHelper(int const newBase, int number, char * result)
257 {
258     int index = 0;
259     while (number != 0)
260     {
261         int currentDigit = number % newBase;
262         result[index] = (char)(currentDigit + '0');
263         number /= newBase;

```

```

264         index++;
265     }
266 }
267
268 /**
269  * @brief A Power operator. Raises the base in the power of degree.
270  * @param base The base of the power.
271  * @param degree The degree of the power.
272  * @return The result of the base raised to the degree.
273  */
274 int power(int const base, int const degree)
275 {
276     if (degree == 0)
277     {
278         return 1;
279     }
280     else
281     {
282         return (power(base, degree - 1)) * base;
283     }
284 }
285
286
287 /*----- Input Handling -----*/
288
289
290 /**
291  * @brief Verify that the given number in the user input can be represented in the
292  *        given original base.
293  * @param originalBase The given original base in the user input.
294  * @param number The given number in the user input.
295  * @return 0 if the input is invalid, 1 otherwise.
296  */
297 int checkInput(int const originalBase, int number)
298 {
299     while (number != 0)
300     {
301         int currentDigit = number % STANDARD_BASE;
302         if (currentDigit >= originalBase)
303         {
304             return FALSE;
305         }
306         number /= STANDARD_BASE;
307     }
308     return TRUE;
309 }
310
311
312 /*----- Output Handling -----*/
313
314
315 /**
316  * @brief Prints the given conversion result to the standard output.
317  *        During the conversion, the result is stored backwards, so this function determines which
318  *        index is the last index in the result array that contains data, and from this points it
319  *        prints the data all the way back.
320  * @param The result of the converted number.
321  */
322 void printResult(char * result)
323 {
324     // Determine the indices of result that contain data.
325     int i = 0;
326     while (result[i] != 0)
327     {
328         i++;
329     }
330
331     // Prints the converted number in the required order.

```

```
332     i--; // index 'i' is currently at the '\0' character, we need to take 1 step backwards.
333     for ( ; i >= 0; --i)
334     {
335         printf("%c", result[i]);
336     }
337     printf("\n");
338 }
```



## 3 CheckParenthesis.c

```
1  /**
2   * @file CheckParenthesis.c
3   * @author Itai Tagar <itagar>
4   * @version 1.2
5   * @date 09 Aug 2016
6   *
7   * @brief A program that verify text files that satisfies a desired parenthesis structure.
8   *
9   * @section LICENSE
10  * This program is free to use in every operation system.
11  *
12  * @section DESCRIPTION
13  * A program that verify text files that satisfies a desired parenthesis structure.
14  * Input:      A name or a path to a text file.
15  * Process:    Validates input, if the input is valid the program starts to analyze the text file
16  *             for determine if the structure of parenthesis is valid or invalid.
17  *             If the file is invalid the program ends with an error message.
18  * Output:     A message that states the file analysis results, if the input was valid.
19  *             An error message in case of bad input.
20  */
21
22
23 /*----- Includes -----*/
24
25
26 #include <stdio.h>
27
28
29 /*----- Definitions -----*/
30
31
32 /**
33  * @def VALID_STATE 0
34  * @brief A Flag for valid state during the program run.
35  */
36 #define VALID_STATE 0
37
38 /**
39  * @def INVALID_STATE 1
40  * @brief A Flag for invalid state during the program run.
41  */
42 #define INVALID_STATE 1
43
44 /**
45  * @def VALID_ARGUMENTS_NUMBER 2
46  * @brief A Macro that sets the valid number of arguments for this program.
47  */
48 #define VALID_ARGUMENTS_NUMBER 2
49
50 /**
51  * @def INVALID_ARGUMENTS_MESSAGE "Please supply a file!\nusage: CheckParenthesis <filename>\n"
52  * @brief A Macro that sets the output message for invalid arguments.
53  */
54 #define INVALID_ARGUMENTS_MESSAGE "Please supply a file!\nusage: CheckParenthesis <filename>\n"
55
56 /**
57  * @def FILE_NAME_INDEX 1
58  * @brief A Macro that sets the index of the File name in the arguments array.
59  */
```

```

60 #define FILE_NAME_INDEX 1
61
62 /**
63  * @def INVALID_FILE_ARGUMENTS_MESSAGE "Error! trying to open the file %s\n"
64  * @brief A Macro that sets the output message for an invalid File argument.
65  */
66 #define INVALID_FILE_ARGUMENTS_MESSAGE "Error! trying to open the file %s\n"
67
68 /**
69  * @def VALID_FILE "ok\n"
70  * @brief A Macro that sets the output message for a valid File.
71  */
72 #define VALID_FILE "ok\n"
73
74 /**
75  * @def INVALID_FILE "bad structure\n"
76  * @brief A Macro that sets the output message for a invalid File.
77  */
78 #define INVALID_FILE "bad structure\n"
79
80 /**
81  * @def INITIAL_SCOPE_NUMBER 0
82  * @brief A Macro that sets the initial scope number in a given File.
83  */
84 #define INITIAL_SCOPE_NUMBER 0
85
86 /**
87  * @def OPEN_ROUND '('
88  * @brief A Flag for the Round Opening-Parenthesis character.
89  */
90 #define OPEN_ROUND '('
91
92 /**
93  * @def CLOSE_ROUND ')'
94  * @brief A Flag for the Round Closing-Parenthesis character.
95  */
96 #define CLOSE_ROUND ')'
97
98 /**
99  * @def OPEN_SQUARE '['
100  * @brief A Flag for the Square Opening-Parenthesis character.
101  */
102 #define OPEN_SQUARE '['
103
104 /**
105  * @def CLOSE_SQUARE ']'
106  * @brief A Flag for the Square Closing-Parenthesis character.
107  */
108 #define CLOSE_SQUARE ']'
109
110 /**
111  * @def OPEN_TRIANGLE '<'
112  * @brief A Flag for the Triangle Opening-Parenthesis character.
113  */
114 #define OPEN_TRIANGLE '<'
115
116 /**
117  * @def CLOSE_TRIANGLE '>'
118  * @brief A Flag for the Triangle Closing-Parenthesis character.
119  */
120 #define CLOSE_TRIANGLE '>'
121
122 /**
123  * @def OPEN_CURLY '{'
124  * @brief A Flag for the Curly Opening-Parenthesis character.
125  */
126 #define OPEN_CURLY '{'
127

```

```

128  /**
129   * @def CLOSE_CURLY '}'
130   * @brief A Flag for the Curly Closing-Parenthesis character.
131   */
132  #define CLOSE_CURLY '}'
133
134
135  /*----- Forward Declarations -----*/
136
137
138  /**
139   * @brief Analyze the results of the 'checkFile' functions, and perform the
140   *         required actions for each scenario.
141   * @param checkFileResult The given result of the 'checkFile' functions.
142   */
143  void analyzeResults(int const checkFileResult);
144
145  /**
146   * @brief Checks the given File for valid parenthesis structure.
147   * @param pFile The given File to check.
148   * @return 0 if the given File satisfies the required parenthesis structure, 1 otherwise.
149   */
150  int checkFile(FILE * const pFile);
151
152  /**
153   * @brief Checks the given File for valid parenthesis structure.
154   *         This function perform recursive calls each time a new parenthesis is opened.
155   * @param currentType The current type of Opening-Parenthesis in this call of the function.
156   * @param pFile The given File to check.
157   * @return 0 if the given File satisfies the required parenthesis structure, 1 otherwise.
158   */
159  int checkFileHelper(char const currentType, FILE * const pFile);
160
161  /**
162   * @brief Checks if a given 2 parenthesis are matching each other (i.e. one closes the other).
163   * @param close The Closing-Parenthesis character.
164   * @param open The Opening-Parenthesis character.
165   * @return 0 if the given 2 parenthesis are matching each other, 1 otherwise.
166   */
167  int checkMatchingParenthesis(char const close, char const open);
168
169
170  /*----- Main -----*/
171
172
173  /**
174   * @brief The main function that runs the program.
175   *         It receives arguments from the user and if the arguments are valid, it runs the File
176   *         Analysis.
177   * @param argc The number of given arguments.
178   * @param argv[] The arguments from the user.
179   * @return 0 if the given File is a text file which satisfies the required
180   *         parenthesis structure, 1 otherwise.
181   */
182  int main(int argc, char * argv[])
183  {
184
185      // Check valid arguments.
186      if (argc != VALID_ARGUMENTS_NUMBER)
187      {
188          fprintf(stderr, INVALID_ARGUMENTS_MESSAGE);
189          return INVALID_STATE;
190      }
191      else
192      {
193          // Receive the File to check.
194          FILE * pFile;
195          pFile = fopen(argv[FILE_NAME_INDEX], "r");

```

```

196
197     // In case of a bad File.
198     if (pFile == 0)
199     {
200         fprintf(stderr, INVALID_FILE_ARGUMENTS_MESSAGE, argv[FILE_NAME_INDEX]);
201         fclose(pFile);
202         return INVALID_STATE;
203     }
204
205     // Analyze the File and close its Stream.
206     int checkFileResult = checkFile(pFile);
207     fclose(pFile);
208
209     // Analyze the results.
210     analyzeResults(checkFileResult);
211     return VALID_STATE;
212 }
213 }
214
215
216 /*----- Analyze File -----*/
217
218
219 /**
220  * @brief Analyze the results of the 'checkFile' functions, and perform the
221  *         required actions for each scenario.
222  * @param checkFileResult The given result of the 'checkFile' functions.
223  */
224 void analyzeResults(int const checkFileResult)
225 {
226     if (!(checkFileResult)) // If the File is valid, 'checkFileResult' will be equal to 0.
227     {
228         printf(VALID_FILE);
229     }
230     else
231     {
232         printf(INVALID_FILE);
233     }
234 }
235
236 /**
237  * @brief Checks the given File for valid parenthesis structure.
238  * @param pFile The given File to check.
239  * @return 0 if the given File satisfies the required parenthesis structure, 1 otherwise.
240  */
241 int checkFile(FILE * const pFile)
242 {
243     return checkFileHelper(EOF, pFile);
244 }
245
246 /**
247  * @brief Checks the given File for valid parenthesis structure.
248  *         This function perform recursive calls each time a new parenthesis is opened.
249  * @param currentType The current type of Opening-Parenthesis in this call of the function.
250  * @param pFile The given File to check.
251  * @return 0 if the given File satisfies the required parenthesis structure, 1 otherwise.
252  */
253 int checkFileHelper(char const currentType, FILE * const pFile)
254 {
255     static int scopeCounter = INITIAL_SCOPE_NUMBER;
256
257     int currentChar; // The current character in the given File.
258
259     while ((currentChar = fgetc(pFile)) != EOF)
260     {
261         // In case we reached any kind of Opening-Parenthesis, we enter a recursive call
262         // and increase the 'scopeCounter' by 1.
263         if (currentChar == OPEN_ROUND)

```

```

264     {
265         scopeCounter++;
266         checkFileHelper(OPEN_ROUND, pFile);
267     }
268     else if (currentChar == OPEN_SQUARE)
269     {
270         scopeCounter++;
271         checkFileHelper(OPEN_SQUARE, pFile);
272     }
273     else if (currentChar == OPEN_TRIANGLE)
274     {
275         scopeCounter++;
276         checkFileHelper(OPEN_TRIANGLE, pFile);
277     }
278     else if (currentChar == OPEN_CURLY)
279     {
280         scopeCounter++;
281         checkFileHelper(OPEN_CURLY, pFile);
282     }
283
284     // In case we reached any kind of Closing-Parenthesis, we determine if it is valid.
285     // If it is valid we exit the current recursive call and decrease the 'scopeCounter' by 1.
286     // If it is invalid, we exit the recursive call with the value 1.
287     if (currentChar == CLOSE_ROUND || currentChar == CLOSE_SQUARE ||
288         currentChar == CLOSE_TRIANGLE || currentChar == CLOSE_CURLY)
289     {
290         if (!(checkMatchingParenthesis((char) currentChar, currentType)))
291         {
292             scopeCounter--;
293             return VALID_STATE;
294         }
295         else
296         {
297             return INVALID_STATE;
298         }
299     }
300 }
301
302 // In case we reached the end of the File, we check that there are no Opening-Parenthesis
303 // left unclosed, using the 'scopeCounter'.
304 if (scopeCounter == INITIAL_SCOPE_NUMBER)
305 {
306     return VALID_STATE;
307 }
308 else
309 {
310     return INVALID_STATE;
311 }
312 }
313
314 /**
315  * @brief Checks if a given 2 parenthesis are matching each other (i.e. one closes the other).
316  * @param close The Closing-Parenthesis character.
317  * @param open The Opening-Parenthesis character.
318  * @return 0 if the given 2 parenthesis are matching each other, 1 otherwise.
319  */
320 int checkMatchingParenthesis(char const close, char const open)
321 {
322     switch (close)
323     {
324         case (CLOSE_ROUND):
325             if (open != OPEN_ROUND)
326             {
327                 return INVALID_STATE;
328             }
329             else
330             {
331                 return VALID_STATE;

```

```

332     }
333
334     case (CLOSE_SQUARE):
335         if (open != OPEN_SQUARE)
336         {
337             return INVALID_STATE;
338         }
339         else
340         {
341             return VALID_STATE;
342         }
343
344     case (CLOSE_TRIANGLE):
345         if (open != OPEN_TRIANGLE)
346         {
347             return INVALID_STATE;
348         }
349         else
350         {
351             return VALID_STATE;
352         }
353
354     case (CLOSE_CURLY):
355         if (open != OPEN_CURLY)
356         {
357             return INVALID_STATE;
358         }
359         else
360         {
361             return VALID_STATE;
362         }
363
364     default:
365         return INVALID_STATE;
366 }
367 }
```