```c


//                    *****   Resolução: E-fólio B   *****

//-------------------------------------------------------------------------------------
//                           Alínea A - gerar a semente
//-------------------------------------------------------------------------------------

/*
*   Nesta alínea houve quem convertesse o número para string em binário,
*   mas pretendia-se que utilizassem apenas operações binárias.
*/

#define MAXSTR 1024
unsigned int GerarSemente(char *str) {
    unsigned int semente = 0;
    while (str[0] != 0) {
        semente += str[0];
        semente ^= (semente << 26) ^ (semente << 15) ^ (semente << 7);
        str++;
    }
    return semente;
}
int main()
{
    char str[MAXSTR];
    if (fgets(str, MAXSTR, stdin) != EOF)
        printf("%u", GerarSemente(str));
}


//-------------------------------------------------------------------------------------
//                  Alínea B - extrair os caracteres codificáveis
//-------------------------------------------------------------------------------------

/*
*   Nesta alínea a versão mais simples, para ficar por aqui,
*   bastaria algo deste tipo:
*/

#define MAXSTR 1024
#define CODIFICAVEIS "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz
\t"
int CharCod(char caracter) {
    return strchr(CODIFICAVEIS, caracter) != NULL;
}
int main() {
    char str[MAXSTR];
    int i;
    while (!feof(f))
        if (fgets(str, MAXSTR, f) != NULL)
            for (i = 0; str[i] != 0; i++)
                if (CharCod(str[i]))
                    printf("%c", texto[i]);
    return 0;
}

/*  Fazia mesmo assim sentido uma função para saber se o caracter é ou não
codificável.
*   A função strchr é a mais genérica, já que podem editar a lita de caracteres
codificáveis.
*   Mas utilizaram várias alternativas mediante os caracteres concretos:
*/

if(ascii == 32 || (ascii >= 48 && ascii <= 57) || (ascii >= 65 && ascii <= 90)||
(ascii >= 97 && ascii <= 122) ...
if((chave[i] >= '0' && chave[i] <= '9') || (chave[i] >= 'a' && chave[i] <= 'z') ||
(chave[i] >= 'A' && chave[i] <= 'Z') || chave[i] == ' ' || chave[i] == '\t')  ...
if(c == '\t' || c == ' ' || isalnum(c)) ...

/*  Naturalmente que a última versão é mais apelativa, mas não há aqui a
```

```
    flexibilidade
67  *   para permitir alterar a string de caracteres codificáveis, mantendo o código
    operacional.
68  *
69  *   Esta versão de código de pouco acrescenta para as alíneas seguintes,
70  *   já que não define função nenhuma. O código se for reutilizado,
71  *   tem de ser sempre modificado, pelo que é uma alternativa pouco genérica.
72  *   Uma melhor alternativa mas mais complexa, é prever já que será necessário
73  *   extrair os caracteres codificáveis de um ficheiro, carregando já o ficheiro para
    memória:
74  */
75
76  #define MAXSTR 1024
77  #define CODIFICAVEIS "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz
    \t"
78  int CharCod(char caracter) {
79      return strchr(CODIFICAVEIS, caracter) != NULL;
80  }
81
82  // função das Formação 3
83  char *Concatenar(char *str, char *str2)
84  {
85      char *pt;
86      /* duplicar str2 se str é nulo */
87      if (str == NULL) {
88          pt = (void *)malloc(strlen(str2) + 1);
89          if (pt != NULL)
90              strcpy(pt, str2);
91      }
92      else {
93          pt = (void *)malloc(strlen(str) + strlen(str2) + 1);
94          if (pt != NULL)
95          {
96              strcpy(pt, str);
97              strcat(pt, str2);
98              free(str);
99          }
100     }
101     return pt;
102 }
103 char *LerFicheiro(char *nome)
104 {
105     char str[MAXSTR];
106     char *texto=NULL;
107     FILE *f=stdin;
108     /* abrir o ficheiro em modo de texto*/
109     if (nome[0] != 0) {
110         f = fopen(nome, "rt");
111         if (f == NULL) {
112             printf("Ficheiro %s não existe.\n", nome);
113             return NULL;
114         }
115     }
116     while (!feof(f))
117         if (fgets(str, MAXSTR, f) != NULL)
118             texto = Concatenar(texto, str);
119     return texto;
120 }
121
122 int main() {
123     char *texto;
124     int i;
125     texto=LerFicheiro("");
126     if (texto != NULL) {
127         for (i = 0; texto[i] != 0; i++)
128             if (CharCod(texto[i]))
129                 printf("%c", texto[i]);
130     }
131     free(texto);
132     return 0;
133 }
```

```
134
135 /*  A função concatenar da formação 3 é suficiente para carregar qualquer texto.
136 *   O argumento no método LerFicheiro é para prever a utilização de um ficheiro,
137 *   em vez de ler o texto através do stdin, essencial para o HR.
138 *   Ficava feito este ponto, já que depois na alínea D haverá outras preocupações.
139 */
140
141 //-------------------------------------------------------------------------------
142 //         Alínea C - codificar uma string de caracteres codificáveis
143 //-------------------------------------------------------------------------------
144
145 /*
146 *   Nesta alínea assume-se que todos os caracteres são codificáveis, ou seja,
147 *   o código será chamado após uma função que faça precisamente o que a alínea B
148 *   executa, portanto não temos que nos preocupar com os caracteres que não são
149 *   codificáveis. Aqui é interessante uma função para codificar uma string,
150 *   de modo a ser uma função com potencial de ser reutilizada em outros contextos.
151 *   A função rodar caracter, é uma função simples, mas mesmo que tivesse uma dimensão
152 *   muito pequena, seria uma função a considerar já que existe possibilidade de se
153 *   querer fazer evoluir o método neste ponto.
154 *
155 *   O código novo será o seguinte:
156 */
157
158 #define TAM_COD 64
159
160 // função das AFs
161 void Baralhar(int v[], int n)
162 {
163     int i, j, aux;
164     /* processar todos os elementos */
165     for (i = 0; i<n - 1; i++) {
166         /* gerar um valor aleatório para sortear o elemento do
167         vector a ficar na posição i (entre i e n-1). */
168         j = i + randaux() % (n - i);
169         aux = v[i];
170         v[i] = v[j];
171         v[j] = aux;
172     }
173 }
174
175 char RodarCaracter(char caracter, int valor)
176 {
177     static char *chCodificaveis = CODIFICAVEIS;
178     char *pt;
179     int indice;
180     pt = strchr(chCodificaveis, caracter);
181     if (pt != NULL) {
182         indice = (int)(pt - chCodificaveis);
183         indice = (indice + valor) % TAM_COD;
184         return chCodificaveis[indice];
185     }
186     return caracter;
187 }
188 void Codificar(char *codificaveis)
189 {
190     char *codificado;
191     int i = 0, tamanho, *indice;
192     tamanho = strlen(codificaveis);
193     indice = (int*)malloc(sizeof(int)*tamanho * 2);
194     if (indice == NULL)
195         return;
196     // inicializar o índice
197     for (i = 0; i < tamanho; i++) // primeira parte a identidade
198         indice[i] = i;
199     Baralhar(indice, tamanho);
200     for (i = tamanho; i < 2 * tamanho; i++) // segunda parte rodar
201         indice[i] = randaux() % TAM_COD;
202     codificado = (char*)malloc(sizeof(char)*(tamanho + 1));
203     if (codificado != NULL) {
204         for (i = 0; i<tamanho; i++)
```

```
205             codificado[i] = RodarCaracter(codificaveis[indice[i]], indice[i+tamanho]);
206         codificado[i] = 0;
207         strcpy(codificaveis, codificado);
208         free(codificado);
209     }
210     free(indice);
211 }
212 // de modo a funcionar no Windows e no Linux (no HR),
213 // tem de se remover o final da string se necessário
214 void RemoverFimLinha(char *str)
215 {
216     int tamanho = strlen(str);
217     while (strchr("\n\r", str[tamanho - 1]) != NULL)
218         str[--tamanho] = 0;
219 }
220 int main() {
221     char str[MAXSTR];
222     fgets(str, MAXSTR, stdin); // lê chave
223     RemoverFimLinha(str);
224     seed=GerarSemente(str);
225     fgets(str, MAXSTR, stdin); // lê caracteres a codificar
226     RemoverFimLinha(str);
227     Codificar(str);
228     printf("%s", str);
229     return 0;
230 }
231
232 /*  Notar no vetor de índices, apenas um vetor para as duas funções, a primeira
233 *   parte é o índice, e a segunda parte é o avanço dos caracteres.
234 *   Mais correto seria dividir o índice em 2, mas como um não existe sem o outro,
235 *   optei por juntar. Neste caso, a alocação de memória e libertação é feita
236 *   na mesma função,garantindo assim que se liberta tudo o que se aloca.
237 *   Pode-se utilizar tipos abstratos de dados para encapsular todas as alocações e
238 *   libertações de memória dentro do tipo abstrato de dados, mas atendendo à
239 *   reduzida dimensão do problema, optei nesta resolução por fazer desta forma.
240 */
241
242
243 //-------------------------------------------------------------------------------
244 //                      Alínea D - codificar/descodificar
245 //-------------------------------------------------------------------------------
246
247 /*
248 *   Nesta alínea houve dificuldades em processar um subconjunto de uma string,
249 *   existindo alguns estudantes com problemas com os caracteres acentuados, que em
250 *   UTF8 em Linux ocupam 2 bytes. Nada mais simples que extrair o subconjunto,
251 *   processar, e depois inserir o subconjunto da string novamente.
252 *   Guardar posições de caracteres ou outras alternativas mais complexas, seria
253 *   razoável se a alínea anterior não tivesse já a funcionar para a situação
254 *   em que se codificam todos os caracteres. Por outro lado, na alínea B
255 *   já extraiam essa substring, pelo que a funcionalidade já estava disponível.
256 *   Como tinham essas funcionalidades implementadas, tinham obrigação de procurar
257 *   reutilizar as funções anteriores, e com o menor número de alterações,
258 *   encontrado a alternativa mais simples: extrair os caracteres a codificar,
259 *   codificar, e inserir novamente os caracteres na string.
260 *   Tinha-se apenas que colocar a alínea B numa função, e fazer a operação inversa,
261 *   com os caracteres codificáveis encriptados, trocá-los na string,
262 *   pelo que mais duas funções seriam úteis para este efeito.
263 */
264
265 // extrair apenas os caracteres codificáveis
266 char * CaracteresCodificaveis(char *texto)
267 {
268     int i, j, tamanho = 1;
269     char *codificaveis;
270     for (i = 0; texto[i] != 0; i++)
271         if (CharCod(texto[i]))
272             tamanho++;
273     codificaveis = (char*)malloc(sizeof(char)*tamanho);
274     if (codificaveis != NULL) {
275         for (i = 0, j = 0; texto[i] != 0; i++)
```

```
276             if (CharCod(texto[i]))
277                 codificaveis[j++] = texto[i];
278         codificaveis[j] = 0;
279     }
280     return codificaveis;
281 }
282 void SubstituirCodificaveis(char *texto, char*codificaveis)
283 {
284     int i, j;
285     for(i=0, j=0; texto[i]!=0; i++)
286         if (CharCod(texto[i]))
287             texto[i] = codificaveis[j++];
288 }
289 void Codificar(char *codificaveis, int cod)
290 {
291     char *codificado;
292     int i = 0, tamanho, *indice, *invertido;
293     tamanho = strlen(codificaveis);
294     indice = (int*)malloc(sizeof(int)*tamanho * 2);
295     if (indice == NULL)
296         return;
297     // inicializar o índice
298     for (i = 0; i < tamanho; i++) // primeira parte a identidade
299         indice[i] = i;
300     Baralhar(indice, tamanho);
301     for (i = tamanho; i < 2 * tamanho; i++) // segunda parte rodar
302         indice[i] = randaux() % TAM_COD;
303     if (!cod) { // inverter os índices, para que o texto seja descodificado
304         invertido = (int*)malloc(sizeof(int)*tamanho);
305         if (invertido != NULL) {
306             // inverter o índice
307             for (i = 0; i < tamanho; i++)
308                 invertido[indice[i]] = i;
309             for (i = 0; i < tamanho; i++)
310                 indice[i] = invertido[i];
311             // inverter o vetor
312             for (i = 0; i < tamanho; i++)
313                 invertido[i] = TAM_COD - indice[indice[i] + tamanho];
314             for (i = 0; i < tamanho; i++)
315                 indice[i + tamanho] = invertido[i];
316             free(invertido);
317         }
318     }
319     codificado = (char*)malloc(sizeof(char)*(tamanho + 1));
320     if (codificado != NULL) {
321         for (i = 0; i<tamanho; i++)
322             codificado[i] = RodarCaracter(codificaveis[indice[i]], indice[i+tamanho]);
323         codificado[i] = 0;
324         strcpy(codificaveis, codificado);
325         free(codificado);
326     }
327     free(indice);
328 }
329
330 int main(int argc, char **argv) {
331     char chave[MAXSTR], cod[MAXSTR]="";
332     char *texto, *codificaveis;
333     int i, contar = 0, tamanho;
334     scanf("%s %s", chave, cod); // lê chave e operação
335     seed = GerarSemente(chave);
336     fgets(chave, MAXSTR, stdin); // lê o resto da linha
337     texto=LerFicheiro("");
338     if (texto == NULL)
339         return 1;
340     codificaveis = CaracteresCodificaveis(texto);
341     if (codificaveis == NULL)
342         return 1;
343     Codificar(codificaveis, strcmp(cod, "codificar") == 0);
344     SubstituirCodificaveis(texto, codificaveis);
345     printf("%s", texto);
346     free(texto);
```

```
347         free(codificaveis);
348         return 0;
349     }
350
351     /*  A função Codificar sofre neste caso apenas uma atualização,
352     *   para inverter os índices. O resto mantém-se igual, pelo que o
353     *   trabalho desta alínea, é apenas a generalização da alínea B,
354     *   acréscimo da alínea C, e a função main.
355     *
356     *   Cumprimentos,
357     *   José Coelho
358     */
359
```