I2181A : Langage C - Modularisation (TP1)

Modularisation

1. Applications avec modules utilisateurs

Le but de cet exercice est de construire deux applications en C :

- une qui « chiffre » une chaîne de caractères et
- une autre qui « déchiffre » une chaîne de caractères.

Une application qui chiffre une chaîne de caractères transforme cette chaîne en une chaîne incompréhensible. Une application de déchiffrement est nécessaire pour transformer la chaîne incompréhensible de manière à retrouver la chaîne compréhensible.

La méthode de chiffrement que nous allons utiliser s'appelle ROT13. Elle est décrite ici:
https://en.wikipedia.org/wiki/ROT13.

Fichier d'en-tête

Nous vous donnons dans le fichier `crypt.c` l'implémentation des deux fonctions de cryptage suivantes :

1. `encrypt`, de type `char*`, qui prend une chaîne de caractères en paramètre et renvoie la chaîne cryptée.
2. `decrypt`, de type `char*`, qui prend une chaîne de caractères en paramètre et renvoie la chaîne décryptée.

Dans un premier temps, nous vous demandons de définir un fichier d'entête `crypt.h` qui **déclare** et **spécifie** ces deux fonctions.

Fichiers sources

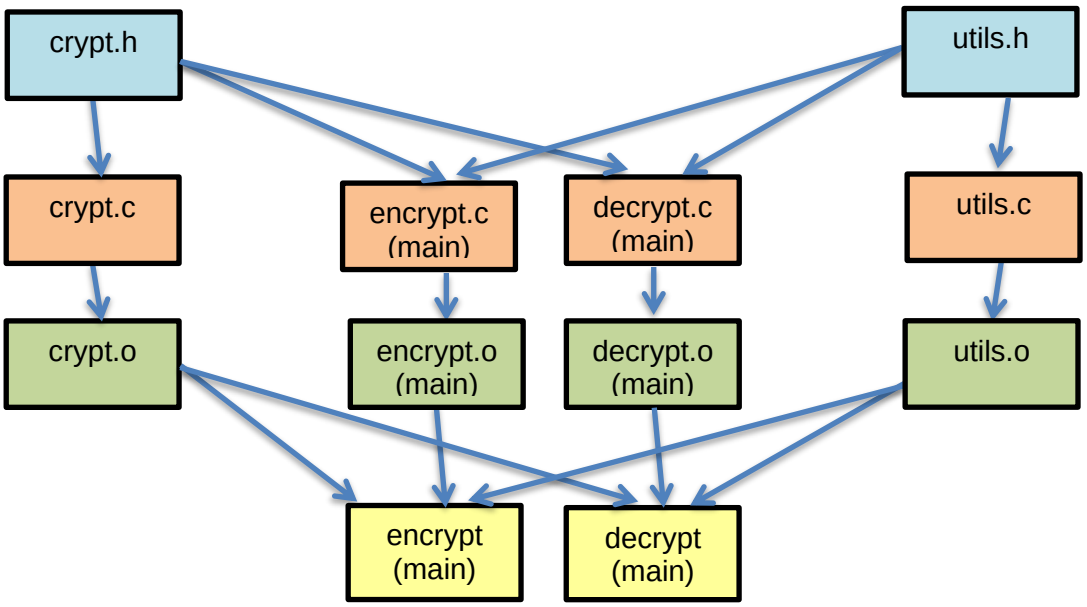
Dans un deuxième temps, nous vous demandons de construire les deux applications suivantes :

1. L'application « `encrypt.c` » qui lit une chaîne de caractères au clavier et qui affiche sur `stdout` la version chiffrée de cette chaîne.
2. L'application « `decrypt.c` » qui lit une chaîne de caractères chiffrée au clavier et qui affiche sur `stdout` la version non-chiffrée de cette chaîne.

Il vous est demandé d'utiliser la fonction de lecture *`readLine()`* fournie dans le module `utils.h` pour lire les chaînes à en/décrypter.

Dépendances

Nous pouvons représenter graphiquement les dépendances entre les différents fichiers de la façon suivante :



Remarquez que les fichiers *.o* sont dépendants de leur source *.c* correspondante mais également par transitivité des entêtes *.h* qu'ils incluent.

Makefile

Dans un troisième temps, nous vous demandons de définir un fichier « `makefile` » qui permet de générer/traiter les cibles suivantes :

- `crypt.o`
- `encrypt.o`
- `decrypt.o`
- `utils.o`
- `encrypt`
- `decrypt`
- `all`
- `clean`

Les cibles *encrypt* et *decrypt* généreront chacune un fichier exécutable du même nom.

Pour chacune de ces cibles, vous devez définir leurs dépendances et la commande qui permet de la générer. Testez chacune de ces règles.

2. *Ajout d'un module utilisateur*

Une seconde manière de crypter une chaîne de caractères vous est demandée. Elle utilise le carré de Polybe (cf. https://fr.wikipedia.org/wiki/Carr%C3%A9_de_Polybe) ci-dessous :

	1	2	3	4	5	6	7	8
1	A	B	C	D	E	F	G	H
2	I	J	K	L	M	N	O	P
3	Q	R	S	T	U	V	W	X
4	Y	Z	0	1	2	3	4	5
5	6	7	8	9		!	"	#
6	$	%	&	‘	(	)	*	+
7	,	-	.	/	:	;	<	=
8	>	?	@	[	\	]	^	_

Pour cette méthode Polybe, nous supposerons que les phrases à crypter ne sont composées que des caractères présents dans le carré ci-dessus.

Nous vous demandons de construire une seconde implémentation du module de cryptage, relative à la méthode du carré de Polybe : « `cryptPolybe.h` » et « `cryptPolybe.c` ». Pour ce faire, mettez à jour votre *Makefile* du 1er exercice afin qu'il génère également les fichiers suivants :

- `cryptPolybe.o`
- `encryptPolybe.o`
- `decryptPolybe.o`
- `encryptPolybe`
- `decryptPolybe`

Pour vous faciliter l'encodage du carré de Polybe, vous pouvez copier la définition de variable globale suivante dans votre code :

```
static char square[8][8] =
 {{'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'},
  {'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P'},
  {'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X'},
  {'Y', 'Z', '0', '1', '2', '3', '4', '5'},
  {'6', '7', '8', '9', ' ', '!', '"', '#'},
  {'$', '%', '&', '\'', '(', ')', '*', '+'},
  {',', '-', '.', '/', ':', ';', '<', '='},
  {'>', '?', '@', '[', '\\', ']', '^', '_'}};
```

3. *Application avec modules utilisateurs*

Nous vous demandons de concevoir un programme *secretAgent* un peu plus complexe que les précédents. Il permet de crypter ou décrypter des phrases selon différents codes.

Voici à quoi pourrait ressembler le menu principal :

```
1. Cryptage
2. Décryptage
Entrez votre choix (Ctrl-D pour terminer) :
```

Une fois l'opération sélectionnée, un second menu propose différents codes. Par exemple :

```
1. ROT13
2. Polybe
3. Morse
Entrez votre choix :
```

Pour finir, une phrase est demandée à l'utilisateur et son (dé)cryptage selon le code sélectionné est affiché à l'écran.

Ce traitement se répète jusqu'à ce que l'utilisateur entre Ctrl-D (EOF).

Reprenez les modules des exercices précédents : ***crypt*** pour le codage ROT13 et ***cryptPolybe*** pour le carré de Polibe.

Remarquez que nous vous proposons d'implémenter un 3e code : le morse. Le code morse international pour être plus précis. Nous nous limiterons aux lettres (pas de chiffre) :

A	● ▬	N	▬ ●
B	▬ ● ● ●	O	▬ ▬ ▬
C	▬ ● ▬ ●	P	● ▬ ▬ ●
D	▬ ● ●	Q	▬ ▬ ● ▬
E	●	R	● ▬ ●
F	● ● ▬ ●	S	● ● ●
G	▬ ▬ ●	T	▬
H	● ● ● ●	U	● ● ▬
I	● ●	V	● ● ● ▬
J	● ▬ ▬ ▬	W	● ▬ ▬
K	▬ ● ▬	X	▬ ● ● ▬
L	● ▬ ● ●	Y	▬ ● ▬ ▬
M	▬ ▬	Z	▬ ▬ ● ●

Le morse est composé de *points* et de *traits*. Les lettres sont séparées par 1 espace et les mots par 3 espaces.

Ex : SOS → `... --- ...`

Implémentez un module ***cryptMorse*** sur le même modèle que les deux modules précédents et créez un nouveau fichier Makefile qui génèrera uniquement l'exécutable *secretAgent*.

Dans un premier temps, n'implémentez que la fonction de codage.

Pour tester votre fonction de décryptage morse, essayez de traduire la phrase du fichier *message_en_morse.txt*.