On travaillera depuis la machine virtuelle, et on compilera les fichiers OCaml et C écrits avant d'exécuter les binaires produits.

Ex.1 Réimplémentation en C de la commande `echo` - C (20 minutes)

On travaillera dans un fichier `echo.c`, important `stdio.h` (pour `printf`).

1. Écrire une fonction `void print_each(int argc, char* array_of_char[])` qui à l'aide de `void printf(string, ...)` affiche chaque argument de `print_each`, *sauf le premier*, chacun séparés par une espace, et avec un retour à la ligne à la fin. Avant la fin de ligne, on fera attention à ne pas afficher une espace qui ne devrait pas être là.
2. Écrire une fonction `int main(int argc, char* argv[])` qui affiche à l'écran tous les arguments du `argv` (*sauf le premier*). On utilisera évidemment `print_each`.
3. Compiler le fichier en un binaire `echo.exe` et vérifier qu'il fonctionne comme la commande `echo` du terminal.

On rappelle qu'on compile ce fichier avec `COMPILATEUR = gcc` ou `clang` avec la ligne de commande suivante, puis on exécute le binaire produit avec la deuxième ligne (sans les dollars qui représentent le prompt de la ligne de commande du terminal) :

```
$ COMPILATEUR -O0 -Wall -Wextra -Wvla -Werror -fsanitize=address
  -fsanitize=undefined -pedantic -std=c11 -o echo.exe echo.c
$ ./echo.exe arg1 arg2 ... argN
arg1 arg2 ... argN
$ echo arg1 arg2 ... argN
arg1 arg2 ... argN
```

Ex.2 Réimplémentation en OCaml de la commande `cat` et un quine - OCaml (35 minutes)

On travaillera dans le fichier `cat.ml`. On rappelle que l'on peut le compiler avec `COMPILATEUR = ocamlc` ou `= ocamlopt` et la ligne de commande suivante, et ensuite que la seconde ligne permet de l'exécuter :

```
$ COMPILATEUR -o cat.exe cat.ml
$ ./cat.exe cat.ml
(* Solution cat.ml TP 11 semaine 10/01/2022 *)

let affiche_fichier nom_fichier =
...
```

1. Écrire une fonction `affiche_fichier: string -> unit` qui ouvre avec `let fichier_in = open_in nom_fichier` le fichier nommé `nom_fichier`, puis qui affiche à l'écran son contenu, ligne par ligne, avec une boucle `while` sur un `let ligne = input_line fichier_in` et `Printf.printf "%s\n" ligne` (ou `print_endline ligne`). On pensera par contre bien

à fermer les flux de fichiers ouverts avec `close_in fichier_in`. Contrairement au C, on n'aura pas besoin de veiller à gérer les erreurs possibles lors de l'ouverture ou la fermeture du fichier, qui déclenchent des exceptions en OCaml contrairement au C qui renvoie des valeurs spéciales (`NULL` pour `fopen` si l'ouverture du fichier échoue, `EOF` si la lecture d'un caractère ou la fermeture échouent). Par contre, pour la lecture d'une ligne, `input_line` renvoie l'exception `End_of_file` lorsqu'elle a atteint la fin du fichier. On utilisera la syntaxe suivante pour capturer l'exception :

```
try begin
    let ligne = input_line fichier_in in
    (* code pour afficher ligne à l'écran *)
end with End_of_file ->
    (* code pour terminer la fonction *)
```

2. Écrire une fonction `main: unit -> unit` qui appelle `affiche_fichier(Sys.argv.(i))` sur chaque argument *sauf le premier* du tableau `Sys.argv`. Si un de ces appels résulte en une erreur, l'exception sera propagée sans être capturée. On pensera bien à appeler la fonction `main()` avant de terminer le fichier, évidemment.
3. Compiler le fichier en un binaire `cat.exe` et vérifier qu'il fonctionne comme la commande `cat` lorsqu'il est exécuté avec un nom de fichier comme argument. On pourra créer un petit fichier `petit_fichier.txt` contenant quelques lignes pour l'exemple.

```
$ cat petit_fichier.txt
Lycée
Kléber
$ ./cat.exe petit_fichier.txt
Lycée
Kléber
```

4. Observez le comportement de `cat` sur un fichier qui n'existe pas. Et comment se comporte votre `cat.exe` ? Est-ce nécessaire de changer son comportement ou bien l'éventuel message d'erreur est déjà assez clair ?
5. Copier-coller le fichier `cat.ml` précédent dans un fichier `quine.ml` et modifier là où c'est nécessaire (dans la fonction `main`) pour faire en sorte que l'exécution du binaire `quine.exe` affiche le code source de `quine.ml`, sans besoin d'argument en ligne de commande :

```
$ ./quine.exe
(* Solution quine.ml TP 11 semaine 10/01/2022 *)

let affiche_fichier nom_fichier =
...
```

Ex.3 Codage de César en C - C (30 minutes)

Écrire une fichier C `codage_cesar.c` important `stdio.h` (pour `printf` et `scanf`), `stdlib.h` (pour `EXIT_SUCCESS` et `EXIT_FAILURE`), et `string.h` pour `strcpy` qui permet la copie d'une chaîne dans une autre.

On rappelle qu'on compile ce fichier avec `COMPILATEUR = gcc` ou `clang` avec la ligne de commande suivante, puis on exécute le binaire produit avec la deuxième ligne (sans les dollars qui représentent le prompt de la ligne de commande du terminal) :

```
$ COMPILATEUR -O0 -Wall -Wextra -Wvla -Werror -fsanitize=address
  -fsanitize=undefined -pedantic -std=c11 -o codage_cesar.exe codage_cesar.c
$ ./codage_cesar.exe petit_fichier.txt petit_fichier_encrypted.txt
```

1. Écrire une fonction `char encrypt(char c_in, int k)` décalant le caractère `c_in` de `k` dans le code ASCII (en restant bien dans un code ASCII, donc modulo 256). Exemple :

```
char c_in = 'm';
char c_out;
c_out = encrypt(c_in, 3);
printf("%c", c_out); // doit afficher p
```

2. Écrire une fonction `int encrypt_file(char* filename_in, char* filename_out, int k)` chiffrant le contenu du fichier de chemin `filename_in`, en écrivant dans un fichier de chemin `filename_out`, en utilisant le codage de César d'un décalage de `k`.

Il faut donc :

— Lire le contenu du fichier `fp_in` caractère par caractère (qui peut donc être un saut de ligne ou une espace ou un caractère imprimable à l'écran), avec `fscanf(fp_in, "%c", c);`.
— Appliquer `encrypt` sur le caractère `c` pour obtenir `c_encrypted` un autre caractère.
— Écrire le caractère `c_encrypted` chiffré dans le fichier `fp_out`.

On encodera caractère par caractère, avec un squelette de code qui suit, à compléter :

```
// Solution TP11_codage_cesar.c TP 11 semaine 10/01/2022
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <assert.h>

// encrypte le char s_in et le renvoie
// en utilisant le codage de César d'un décalage de +k (modulo 256)
char encrypt(char c_in, int k) {
    return TODO;
}

int encrypt_file(const char* filename_in, const char* filename_out, int k) {
    FILE* fp_in = fopen(filename_in, "r");
    if(fp_in == NULL) { // si f = NULL, il y a eu une erreur
        fprintf(stderr, "Le fichier \"%s\" n'a pas pu être ouvert.", filename_in);
        return EXIT_FAILURE;
    }
    FILE* fp_out = fopen(filename_out, "w");
```



```
    if(fp_out == NULL) { ... }

    char c_in;
    char c_encrypted;
    int code_retour_lecture;
    int code_retour_ecriture;
    while (true) {
        code_retour_lecture = fscanf(fp_in, "%c", &c_in); // lit un caractère, stockés d
        // à compléter, pour encoder le char c_in en c_encrypted, et écrire c_encrypted
        ...

        if (code_retour_lecture == EOF) {
            if (feof(fp_in)) { // fin du fichier, erreur normale
                return EXIT_SUCCESS;
            } else {
                return EXIT_FAILURE; // autre erreur
            }
        }

        if (code_retour_ecriture == EOF) {
            ...
        }
    }
    if (fclose(fp_in) == EOF) { ... }
    if (fclose(fp_out) == EOF) { ... }
    return EXIT_SUCCESS;
}

int main(int argc, char* argv[]) {
    assert (argc >= 4);
    return ...;
}
```

3. Écrire une fonction `int main(int argc, char* argv[])` qui prenne trois arguments : un nom de fichier à encrypter, le nom de sa destination, et le décalage `k` que l'on convertira d'une chaîne à un entier avec la fonction `atoi(argv[3])` (qui vient de `stdlib`). On écrira un fichier `TP11_codage_cesar.c` et on le testera sur un petit fichier avec un petit décalage par exemple `k=5`.
4. (Bonus) Écrire une fonction `decrypt` qui fasse l'inverse de `encrypt`, et ensuite un programme entier `TP11_decodage_cesar.c` qui fasse le codage inverse. Vérifiez que vous pouvez décoder le fichier test utilisé pour tester le binaire produit à la question précédente et que vous retrouvez bien le même fichier texte.

Ex.4 Un mini clone de `grep` - OCaml (30 minutes)

On cherche à écrire un programme `grep.ml` qui une fois compilé en `grep.exe` accepte comme arguments en ligne de commande deux arguments `mot` et `fichier_in`, et affiche toutes les lignes du fichier `fichier_in` qui contiennent ce mot `mot`.

1. Écrire une fonction `contient (mot : string) (ligne : string) -> bool` qui teste avec l'algorithme le plus naïf qui soit si la (courte) chaîne `mot` est présente dans la (longue) chaîne `ligne`. On pensera à utiliser une boucle `for`, la fonction `String.length` pour obtenir les longueurs des chaînes, et `String.sub ligne debut longueur` qui lit une sous-chaîne dans une grande chaîne. La comparaison de chaîne se fait en OCaml avec le simple opérateur `=`, inutile d'appeler une fonction spéciale comme en C (où il faut appeler `strcmp`).
2. Écrire une fonction `contient_fichier (mot : string) (chemin_in : string) -> unit` qui ouvre le fichier de chemin `chemin_in` avec `open_in`, qui lit chacune de ses lignes jusqu'à tomber sur une exception `End_of_file` (que l'on rattrapera), et affiche à l'écran chaque ligne qui contiennent le mot `mot` donné.
3. Écrire une fonction `main : unit -> unit` qui appelle la fonction précédente avec les deux premiers arguments donnés en ligne de commande (donc présents dans le tableau de string `Sys.argv` aux indices `1` et `2`).

```
$ cat fichier_test_grep.txt
1 Ligne avec le mot test
2 Ligne sans le mot
3 Ligne avec le mot test
4 Ligne sans le mot

$ grep test fichier_test_grep.txt        # le binaire du système
1 Ligne avec le mot test
3 Ligne avec le mot test

$ ./grep.exe test fichier_test_grep.txt  # votre binaire
1 Ligne avec le mot test
3 Ligne avec le mot test
```

