Développement C **L2 CUPGE 2020–2021**

Travaux pratiques EntréesSorties

Dans ce thème, nous verrons comment un programme écrit en C interagit avec son environnement, que ce soit un terminal (la fenêtre dans laquelle il est exécuté) ou des fichiers.

I Exercice : *Déclarations de variables*

À savoir : Une variable est définie par son type (qui est le type des valeurs qu'elle peut contenir) et son adresse dans la mémoire. Quelques règles pour l'utilisation de variables :

1. toute variable doit être *déclarée* dans une fonction ;
2. Le compilateur assure que toute variable déclarée est définie (a une case correspondante dans la mémoire) de manière unique pour chaque utilisation de la fonction ;

   En pratique, on ne fait pas de différence entre une variable et la case dans la mémoire qui lui est associée.

3. une variable est déclarée par un nom et un type :

   Exemple 1 :
   - `int x` déclare une variable de nom `x` et de type `int`
   - `char c` déclare une variable de nom `c` et de type `char`

4. Pour les déclarations complexes, on écrit les opérations qu'il faut faire sur la variable pour obtenir le type au début de la déclaration :

   Exemple 2 : `char * argv[]` signifie que `* argv[i]` (pour un entier `i`) est de type `char` ;
   Donc `argv[i]` est l'adresse d'une case qui contient un `char`
   Donc `argv` est un tableau dont les cases contiennent des adresses de cases de type `char`.

(a) Dans le répertoire ~/ProgrammationS3/TP/src/Types, ouvrir un nouveau fichier `exercice1.1.c`, et écrire un programme minimal.

```
#include <stdio.h>
int
main ( int argc , char * argv[ ] )
{
  return 0 ;
}
```

(b) Compilez et exécutez le programme : dans le terminal, allez dans le répertoire ~/ProgrammationS3/TP/src/Types. Le *compilateur* `gcc` prend comme argument un fichier `toto.c`, et va créer un autre fichier, `a.out`, qui sera la version du programme préparée pour être exécutée sur l'ordinateur. Pour exécuter un programme, il faut taper le chemin vers le programme. Si vous n'êtes pas dans le bon répertoire, les commandes à taper sont :

Code bash 1 – Terminal

```
cd ~/ProgrammationS3/TP/src/Types
# Vérifiez que le fichier exercice1.c existe
ls
# Si c'est le cas, vous êtes dans le bon répertoire
gcc exercice1.c
# Vérifiez que le fichier a.out a été créé
ls
# Exécutez le fichier a.out
./a.out
# il ne se passe rien, normal.
```

(c) Sauvegardez le programme dans le fichier `exercice1.2.c`. Modifiez ce fichier pour afficher la valeur de `argc` :

```
#include <stdio.h>
int
main ( int argc , char * argv[ ] )
{
      // Modifiez le contenu des guillemets
      // comme vous voulez. Il faut juste
      // avoir quelque part %d pour afficher
      // l'entier dont la valeur est mise après
      // les guillements. On peut aussi utiliser %x
      // pour afficher la valeur en hexadécimal.
      printf ( "argc vaut %d\n" , argc ) ;
      return 0 ;
    }
```

(d) Compilez le programme en le fichier ~/ProgrammationS3/TP/bin/exerciceTypes1.2, et exécutez le :

Code bash 2 – Terminal

```
# l'option -o de gcc permet de préciser un autre nom
# que a.out pour le fichier créé
gcc exercice1.2.c -o ../../bin/exerciceTypes1.2
../../bin/exerciceTypes1.2
```

(e) C'est fatigant de devoir toujours écrire ../../bin/. Le terminal a une variable PATH qui contient une liste de répertoires pour lesquels il suffit de donner le nom du fichier qu'on veut exécuté (par exemple, le programme `gcc` est normalement dans /usr/bin, ce que vous pouvez vérifier avec la commande `which gcc`). Pour ajouter le répertoire ~/ProgrammationS3/TP/bin, vous pouvez taper les commandes suivantes. **Attention, vérifiez plusieurs fois la commande, si vous la tapez mal, vous pouvez rendre votre compte inutilisable en effaçant votre configuration.**

Code bash 3 – Terminal

```
# modification de la configuration
echo "export PATH=~/ProgrammationS3/TP/bin:\$PATH" >> ~/.bashrc
# mise à jour de la configuration actuelle
```

```
source ~/.bashrc
```

(f) Exécutez le programme en mettant des mots et des nombres après le nom du programme (sur la même ligne). Que vaut `argc` ?

(g) Les éléments du tableau *argv* sont les mots (séparés par des espaces) sur la ligne de commande. C'est ce mécanisme qui permet à un programme d'avoir des arguments et des options : le programme va lire ce qui a été marqué dans la commande. La fonction `atoi` permet de traduire un mot en un entier. Écrivez un programme `exercice1.3.c` qui affiche en hexadécimal l'entier qui lui est donné en argument. Par exemple, on veut :

Code bash 4 – Terminal

```
exerciceTypes1.3 9
0x9
exerciceTypes1.3 15
0xF
exerciceTypes1.3 123456
0x1E240
```

Normalement, le compilateur ne devrait pas être content, car on a utilisé une fonction qui est incluse dans tous les programmes, mais qui n'a pas été déclarée. Pour qu'il arrête de râler, il faut ajouter au début du fichier :

```
#include <stdlib.h>
```

Notez qu'on veut les lettres en majuscule, pas en miniscules. Pour savoir tout ce qu'on peut faire avec `printf` (environ 400 lignes de documentation, il faut chercher !), vous pouvez utiliser le manuel :

Code bash 5 – Terminal

```
man -S 3 printf
```

II Exercice : *Affichage d'un entier.*

Le 'f' de `printf` signifie *mettre en forme*. Cette fonction est un peu spéciale car contrairement aux fonctions normales en C, elle peut prendre un nombre variable d'arguments, mais il en faut au moins 1.

Le premier argument est une *chaîne de formattage* qui sert à mettre en forme les données qui sont contenues dans les autres arguments, dans l'ordre de lecture. Pour celà, la chaîne de formattage contient des *directives d'affichage*. La première directive sera utilisée pour afficher la première donnée après la chaîne de formattage, la seconde directive sera utilisée pour afficher la seconde donnée après la chaîne de formattage, etc.

Directives d'affichage. Une directive commence toujours par %. Il y en a des dizaines, donc on va juste en regarder quelques unes :

- `%d` : permet d'afficher un entier en base dix ;
- `%c` : permet d'afficher une lettre (un caractère) à partir de son encodage en machine ;
- `%f` : permet d'afficher un nombre à virgule ;
- `%s` : permet d'afficher une chaîne de caractères ;

Pour l'exercice 2, on commence par copier le contenu du programme `exercice1.c` dans un nouveau fichier `exercice2.c`.

(a) Modifiez le premier programme pour afficher sur 10 lignes chaque caractère de '0' à '9' suivie de la valeur entière de l'encodage machine du caractère. Que remarque-t'on ?

(b) Faites de même pour les lettres `'a'` et `'z'`, et affichez leur différence `'z' - 'a'`. Idem pour les majuscules.

III Exercice : *Classes de caractères*

Pré-requis : Thème Variables, Constantes, et Types en C, exercice 1.

Comme les caractères de même nature (lettres minuscules, chiffres, etc.) se suivent, on va écrire un programme qui va lire un caractère, et va afficher sa nature (lorsqu'on la connaît). Par exemple, l'encodage du caractère '5' est bien compris entre l'encodage du caractère '0' et l'encodage du caractère '9'.

IV Exercice : *Chaînes de caractères*

Pré-requis : Thème Variables, Constantes, et Types en C, exercice 3.

Contrairement à Python, C fait la différence entre un caractère seul et une suite de caractères. Les valeurs de type caractère (`char`) s'écrivent entre des apostrophes, comme `'a'`, `'3'`, `'\n'`. Les *chaînes de caractères* sont juste des tableaux de caractères. Pour indiquer la fin de la chaîne, on met dans le tableau l'entier 0, qu'on peut aussi écrire `'\0'`. Il s'agit d'une convention qui est utilisée pour pouvoir traîter plus facilement les chaînes de caractères :

- Il n'y a pas de type spécifique : une chaîne de caractères est définie par l'adresse du premier caractère, donc on utilise `char *` ;
- Pour afficher une chaîne de caractères avec `printf`, on utilise la directive d'affichage `%s` :

Exemple 3 : Les deux morceaux suivants de programme font la même chose :

```
printf ( "Bonjour.\n" ) ;

char * hello = "Bonjour" ; /* Création de la chaîne
                ↪ de caractères */
printf ( "%s.\n" , hello ) ;
```

(a) Écrivez dans le fichier `exercice4.c` un programme qui affiche les 3 premiers éléments du tableau de chaînes de caractères passé en paramètre du main `argv`. Compilez et exécutez ce programme avec *et sans* arguments sur la ligne de commande. Exemple ci-dessous avec 4 arguments (nombre récupéré grâce à argc) qui sont 4 chaînes de caractères qui sont le nom de l'exécutable suivi de 3 arguments sans signification particulière :

```
exercice4 toto 12 argument3
```

Que remarque-t'on ?

(b) Dans le terminal, tapez la commande :

```
echo $LANG
```

Que remarquez vous ?

(c) On va maintenant tenter de voir ce qu'il y a plus loin. Affichez la case 80 du tableau

`argv`.

V Exercice : *Lecture d'entrées*

Même s'il est possible de lire tout ce que tape l'utilisateur avec la fonction `getchar` pour ensuite traduire ce qui est marqué, on utilise en général la fonction `scanf` qui permet de lire des entrées structurées en utilisant des directives.

(a) Écrivez le programme suivant :

```
#include <stdio.h>

int
main ( int argc , char * argv[] )
{
  printf ( "Écrivez: Bonjour, monde !\n" ) ;
  scanf ( "Bonjour, monde !" ) ;
  printf ( "Je vous ai compris!\n" ) ;
  return 0 ;
}
```

Et essayez de vérifier que `scanf` a bien lu ce que vous tapez.

(b) Pour faire la différence entre une lecture réussie et une lecture qui a échoué, lisez et affichez tous les caractères que `scanf` n'a pas réussi à lire jusqu'au caractère de fin de ligne (`'\n'`) en utilisant la fonction `getchar`.

(c) L'intérêt de `scanf` est de pouvoir lire des données avec les types existants. Au contraire de `printf`, il ne faut pas donner en argument une **valeur** mais l'adresse à laquelle il faudra stocker la valeur qui a été lue. Modifiez le programme précédent pour lire une chaîne de la forme "Bonjour, je suis XXX", où XXX est un entier. Le programme devra répondre par "Bonjour, XXX".

VI Exercice : *ROT13 et code de César*

Le ROT13 est un algorithme très simple de chiffrement de texte. Il s'agit de remplacer chaque lettre du texte par son 13$^{\text{ème}}$ successeur dans l'alphabet. Pour les lettres dans la deuxième partie de l'alphabet, on considère que l'alphabet est cyclique. Ainsi la lettre `'a'` est remplacée par la lettre `'n'` et la lettre `'z'` par la lettre `'m'`.

(a) Implémenter un programme qui applique ROT13 au texte tapé par l'utilisateur. Pour lire toute une ligne on utilisera le code suivant :

```
char * ligne;
char   c;
scanf("%m[^\n]%c", &ligne, &c);
// ... Utilisation de ligne ...
free(ligne)
```

Note : la fonction `scanf` va mettre dans `ligne` l'adresse de la première case d'un "tableau" qui restera défini même quand la fonction `scanf` aura terminé. On détruit ce tableau avec la

fonction `free`.

Le programme doit modifier chaque lettre (minuscule ou majuscule) mais laisser les caractères accentués tels quels. Par exemple la chaine "Ceci est le code de César." doit devenir "Prpv rfg yr pbqr qr Péfne.".

Le chiffrement ROT13 est en fait un cas particulier du code de César, qui pour un certain n remplace chaque lettre par son $n^{\text{ème}}$ successeur.

(b) Modifiez le programme précédent pour obtenir un programme encodant le texte tapé par l'utilisateur dans le code de César, le paramètre n étant donné en premier argument du programme.

VII Exercice : *Librairie d'entrées/sorties*

Les exercices précédents ont montré que faire quelque chose d'extrèment simple, comme lire un entier, était en fait très long à écrire proprement. On va maintenant écrire une librairie qui va permettre de lire proprement les types d'entrées les plus fréquents.

(a) Écrire une fonction `lire_fin_ligne` qui lit tous les caractères jusqu'à la fin de la ligne (jusqu'au caractère `'\n'`, donc). Cette fonction renvoie le nombre de caractères lus qui ne sont pas des espaces (pour lesquels la fonction `isspace` définie dans `ctype.h` renvoie vrai).

(b) Écrire une fonction `lire_entier` qui prend en entrée l'adresse d'un entier, et tente de lire jusqu'à ce qu'elle réussisse un entier. Pour que la lecture réussisse, il faut qu'il n'y ait que l'entier et des caractères blancs sur la ligne.

(c) Écrire une fonction `lire_decimal` qui prend en entrée l'adresse d'un décimal, et tente de lire jusqu'à ce qu'elle réussisse un entier. Pour que la lecture réussisse, il faut qu'il n'y ait que le nombre décimal et des caractères blancs sur la ligne.

(d) Créez un fichier include/entrees.h qui contient les déclarations de ces trois fonctions, et compilez le fichier `entrees.c` (on ne peut pas l'exécuter, il n'y a pas de fonction `main`).

(e) Séparez le fichier `entrees.c` en trois fichiers, un pour chaque fonction. Assurez-vous que chaque fichier peut être correctement compilé.

(f) Créez une librairie lib/libentrees.a à partir des 3 fichiers que vous venez d'obtenir par compilation.

(g) On va automatiser la création et la mise à jour de la bibliothèque, car sinon on perd beaucoup de temps à taper des commandes. Pour préparer l'automatisation, il faut d'abord noter (sur une feuille de papier) :

1. quel est le fichier final qu'on veut obtenir ;
2. récursivement, en partant de ce but, et pour chaque fichier à obtenir
   (a) si le fichier est créé par une commande, noter la commande, et ajouter les fichiers qu'utilise cette commande pour produire le fichier voulu ;
   (b) sinon, notez quels fichiers ce fichier utilise (par exemple, quels fichiers `.h`).

(h) On va créer un fichier `entrees.mk` qui va automatiser la création et la mise à jour de la bibliothèque à partir de ces informations. Ce fichier contient des règles, une par fichier à produire. La forme de chaque règle est :

```
fichier à produire : fichiers utilisés
        commande, si nécessaire
```

La seule contrainte est que la première règle doit produire le fichier final.

(i) Recompilez la librairie en utilisant :

```
make -f entrees.mk
```

(j) Écrire une fonction `lit_format` qui prend en entrée une chaîne de caractère (de formattage) et une adresse (de type `void *`). Modifiez la librairie (et le fichier `entrees.h`, et le fichier `entrees.mk`) pour ajouter le code de cette fonction. Modifiez les fonctions `lire_entier` et `lire_decimal` pour qu'elles ne fassent qu'utiliser la fonction `lire_format`.

(k) Ajoutez une nouvelle fonction `lire_entier_positif` qui appelle la fonction `lire_entier` tant que l'entier lu n'est pas positif (strictement).

(l) Créez un programme qui lit n nombres décimaux et affiche leur somme en utilisant la bibliothèque. Créez un fichier *somme.mk* permettant d'automatiser la compilation de ce programme.