Développement C **L2 CUPGE 2018–2019**

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 : *Premier programme*

On va travailler sur les types, donc dans le répertoire ~/workspace/programmation-en-C-CUPGE/src/Entrées-Sorties/.

(a) Vérifiez que vous êtes bien sur la branche `master`, et si ce n'est pas le cas, allez sur cette branche.

Solution

```
git branch
git checkout etudiant
```

(b) Créez le fichier exercice1.c dans le répertoire ~/workspace/programmation-en-C-CUPGE/src/Entrées-Sorties/ et ouvrez-le.

ATTENTION !!! Il ne faut jamais faire du copier-coller d'un fichier pdf vers un fichier texte : même si les caractères affichés se ressemblent, ils sont différents.

(c) Tapez le code suivant dans le fichier :

```
#include <stdio.h>

int main ( int argc , char * argv[] )
{
  return 0;
}
```

(d) Sauvegardez le fichier (avec Ctrl-S).

On va maintenant compiler et exécuter ce programme.

(e) Changer de répertoire pour aller dans ~/workspace/src/Entrées-Sorties.

Solution

```
cd ~/workspace/src/Entrées-Sorties
```

(f) Compiler le programme avec la commande :

Solution

```
gcc -o exercice1 exercice1.c
/* ou bien gcc -o exercice1.exe exercice1.c */
```

(g) Exécutez le programme. Que se passe-t'il ?

Solution

```
./exercice%\arabic{exercicenum}%
```

Il ne se passe rien.

(h) Ajoutez la ligne suivante juste au dessus de `return 0` :

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

et compilez et exécutez le programme.

> `printf` est une *fonction* qui permet à un programme d'envoyer des données au terminal dans lequel il est exécuté.

(i) Ajoutez `\n` après Bonjour, compilez et exécutez le programme.

> `\n` est un caractère qui demande au terminal de passer à la ligne suivante.

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.

Solution

```
#include <stdio.h>

int main ( int argc , char * argv[] )
{
  print ( "%c -> %d \n" , '0' , '0' ) ;
  print ( "%c -> %d \n" , '1' , '1' ) ;
  print ( "%c -> %d \n" , '2' , '2' ) ;
  print ( "%c -> %d \n" , '3' , '3' ) ;
  print ( "%c -> %d \n" , '4' , '4' ) ;
  print ( "%c -> %d \n" , '5' , '5' ) ;
  print ( "%c -> %d \n" , '6' , '6' ) ;
  print ( "%c -> %d \n" , '7' , '7' ) ;
  print ( "%c -> %d \n" , '8' , '8' ) ;
  print ( "%c -> %d \n" , '9' , '9' ) ;
  return 0 ;
}
```

Que remarque-t'on ?

Solution

les codes se suivent dans le même ordre que les chiffres.

(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'.

Solution

```
#include <stdio.h>

int main ( int argc , char * arv[] )
{
  int x ;
  x = getchar ( ) ;
  if ( ( '0' <= x ) && ( x <= '9' ) )
  {
    printf ( "%c est un chiffre.\n" , x ) ;
  }
  if ( ( 'a' <= x ) && ( x <= 'z' ) )
```

```
  {
    printf ( "%c est une lettre minuscule.\n" , x ) ;
  }
  if ( ( 'A' <= x ) && ( x <= 'Z' ) )
  {
    printf ( "%c est une lettre majuscule.\n" , x ) ;
  }
  return 0 ;
}
```

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 1 : 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 ?

Solution

On utilise le programme suivant :

```
#include <stdio.h>

int main ( int argc , char * argv[] )
{
    char * hello = "Bonjour" ;
    printf ( "%s.\n" , hello ) ;
    printf ( "Bonjour.\n") ;
    printf ( "argument 0 : %s\n" , argv[0] ) ;
    printf ( "argument 1 : %s\n" , argv[1] ) ;
```

```
    printf ( "argument␣2 :␣%s\n" , argv[2] ) ;
    printf ( "argument␣3 :␣%s\n" , argv[3] ) ;
    printf ( "argument␣4 :␣%s\n" , argv[4] ) ;
    printf ( "argument␣5 :␣%s\n" , argv[5] ) ;
    printf ( "argument␣6 :␣%s\n" , argv[6] ) ;
    return 0 ;
}
```

Une fois que les arguments sont terminés, il y a une case qui vaut `(null)` (c'est l'adresse 0), suivie de cases qui ont une valeur de la forme `NOM=chaine`

(b) Dans le terminal, tapez la commande :

echo $LANG

Que remarquez vous ?

Solution

Plein de choses !

- La mémoire du programme contient aussi les valeurs des *variables d'environnement* qui sont définies dans le terminal où le programme a été exécuté.
- En fait, la mémoire contient deux tableaux l'un à côté de l'autre :
  - Dans le premier, il y a l'adresse des arguments du programme ;
  - Dans le second, il y a la valeur des variables d'environnement ;
  - La fin de ces tableaux est marquée avec l'adresse 0 (comme la fin d'une chaîne de caractères est marquée par l'entier 0) ;
  - **Très mauvaise nouvelle** : en C, il n'y a aucun mécanisme qui oblige à rester dans un tableau. Quand on parcourt les cases tableau, si on ne fait pas attention, on fini par tomber sur des cases dans la mémoire qui peuvent être définies (ou pas), et qui contiennent d'autres données au bon vouloir du compilateur.

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

Solution

Lors des tests, celà suffit pour avoir un *segmentation fault*. Donc si accède à une case qui n'est pas dans le tableau, on peut avoir soit :

- un résultat qui vient d'ailleurs (de gcc, par exemple) ;
- une erreur qui arrête le programme. *Segmentation fault* signifie qu'on veut accéder à une adresse dans la mémoire qui n'est pas définie.

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 :

```c
#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.

Solution

Pas de différence notable entre le cas où on tape ce qui est demandé, et celui où on tape n'importe quoi.

(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`.

Solution

```c
#include <stdio.h>

int
main ( int argc , char * argv[] )
{
  int c ;
  printf ( "Écrivez : Bonjour , monde ! \n" ) ;
  scanf ( "Bonjour , monde ! " ) ;
  c = getchar () ;
  while ( c != '\n' )
  {
    printf ( "%c" , c ) ;
    c = getchar ( ) ;
  }
  return 0 ;
}
```

Si l'utilisateur a correctement écrit, on n'affiche rien, sinon, le programme affiche les caractères à partir du moment où ça ne marche plus. Notez aussi que les blancs peuvent être ajoutés ou enlevés là où il y en a dans la chaîne de formattage.

(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".

Solution

```
#include <stdio.h>

int
main ( int argc , char * argv[] )
{
  int i ;
  int c ;
  scanf ( "Bonjour, je suis %d" , & i ) ;
  c = getchar () ;
  while ( c != '\n' )
  {
    printf ( "%c" , c ) ;
    c = getchar ( ) ;
  }
  printf ( "Bonjour, %d.\n" , i ) ;
  return 0 ;
}
```

Si l'utilisateur a correctement écrit, on n'affiche rien, sinon, le programme affiche les caractères à partir du moment où ça ne marche plus. Notez aussi que les blancs peuvent être ajoutés ou enlevés là où il y en a dans la chaîne de formattage.

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).

Solution

```
#include <ctype.h>
#include <stdio.h>

int lire_fin_ligne ( )
{
  int count = 0 ;
  int caractere ;
  while ( ( caractere = getchar ( ) ) != EOF )
  {
    if ( caractere == '\n' )
      break ;
    if ( isspace ( caractere ) )
      continue ;
    count++ ;
  }
  return count ;
}
```

(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.

Solution

```
void lire_entier ( int * n)
{
  int lu = 0 ;
  int caractere ;
  do
  {
    lu = scanf ( "%d" , n ) ;
```

```
    nb = lire_fin_ligne ( ) ;
  } while ( ( lu != 1 ) || ( nb > 0 ) ) ;
}
```

(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.

Solution

```
void lire_decimal ( float * n)
{
  int lu = 0 ;
  int caractere ;
  do
  {
    lu = scanf ( "%f" , n ) ;
    nb = lire_fin_ligne ( ) ;
  } while ( ( lu != 1 ) || ( nb > 0 ) ) ;
}
```

Note pour plus tard : La fonction `lire_decimal` ressemble tellement à la fonction `lire_entier` qu'on a très envie de ne l'écrire qu'une seule fois.

(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`).

Solution

```
int lire_fin_ligne ( ) ;
lire_entier ( int * ) ;
lire_decimal ( float * ) ;
```

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

Solution

Il faut inclure le fichier `entrees.h` dans chacun des fichiers `.c`.

```
gcc -c lire_fin_ligne.c -I ../../include
gcc -c lire_entier.c -I ../../include
gcc -c lire_decimal.c -I ../../include
```

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

Solution

```
ar rcs ../../lib/libentrees.a lire_fin_ligne.o lire_entier.o lire_decim
```

(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`).

Solution

1. fichier à produire : `lib/libentrees.h`
   (a) commande :
   ```
   ar rcs ../../lib/libentrees.a lire_fin_ligne.o lire_entier.o
   ```
   (b) fichiers utilisés : lire_fin_ligne.o lire_entier.o lire_decimal.o ;
2. fichiers à produire : lire_fin_ligne.o lire_entier.o lire_decimal.o
   (a) commande (pour chaque fichier) :
   ```
   gcc -c lire_fin_ligne.c -I ../../include
   ```
   (b) fichiers utilisés : lire_fin_ligne.c, lire_entier.c, lire_decimal.c.
3. fichiers à produire : lire_fin_ligne.c lire_entier.c lire_decimal.c
   (a) pas de commande ;
   (b) fichiers utilisés : entrees.h pour chacun des 3 fichiers `.c`

(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.

Solution

```
../../lib/libentrees.a: lire_fin_ligne.o lire_entier.o lire_decimal.o
	ar rcs ../../lib/libentrees.a lire_fin_ligne.o lire_entier.o lire_de

lire_fin_ligne.c: ../../include/entrees.h
lire_entier.c: ../../include/entrees.h
lire_decimal.c: ../../include/entrees.h
```

```
lire_fin_ligne.o: lire_fin_ligne.c
	gcc -c lire_fin_ligne.c -I ../../include

lire_entier.o: lire_entier.c
	gcc -c lire_entier.c -I ../../include

lire_decimal.o: lire_decimal.c
	gcc -c lire_decimal.c -I ../../include
```

(i) Recompilez la librairie en utilisant :

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

Solution

S'il ne fait rien, tout est normal. Modifiez un des fichiers `.c` pour l'obliger à recompiler.

(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.