

Algorithmique et Programmation – Examen (durée 3 heures)

Première session du 9 juin 2021

NOTES : Aucun document autorisé. Sont interdits les calculatrices, les téléphones, ainsi que tout autre ustensile de calcul ou de communication.

REMARQUE 1 : Une indication sur le niveau atteint est précisée après chaque exercice. Le niveau de difficulté allant croissant, il est préférable de traiter les exercices dans l'ordre. La réussite de l'exercice 1 est requise pour valider l'examen ; les exercices peuvent ensuite être traités indépendamment de leur ordre.

REMARQUE 2 : Dans la suite, les indications concernant les nombres de lignes sont données en comptant toute ligne de code non-vide (`#include`, prototypes, accolades...).

REMARQUE 3 : Les petits oublis d'un point-virgule, d'une parenthèse, d'une virgule, d'une apostrophe double ou simple, ne seront pas pénalisants tant qu'ils restent **punctuels**. En revanche, soignez le placement des accolades.

Exercice 1 : Inclusion mutuelle d'objets communicants

Les objets communicants sont de plus en plus nombreux (téléphones et consoles de jeux, casques de vélo et plantes vertes, ...). Nous nous intéressons à l'atteignabilité d'un objet par un autre pour savoir si les deux objets peuvent communiquer par ondes radio, sans nous préoccuper de la technologie utilisée. Pour simplifier, dans le cadre du présent exercice, nous considérons que les récepteurs ont tous la même sensibilité, qu'il n'y a pas d'obstacles entre émetteurs et récepteurs, et que la portée d'un émetteur est la distance maximum à laquelle un récepteur pourra réceptionner et démoduler le signal.

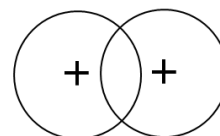
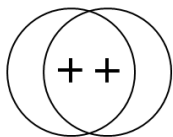
Cet exercice cherche à initier une cartographie d'objets communicants afin de pouvoir calculer les communications possibles.

a) Pour commencer, écrire une **FONCTION** en **LANGAGE C** qui prend en paramètre :

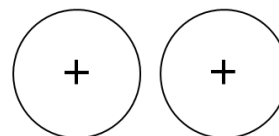
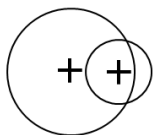
- `c1x`, `c1y`, `c1r` : le centre et le rayon d'un premier cercle ;
- `c2x`, `c2y`, `c2r` : le centre et le rayon d'un deuxième cercle ;

et retourne :

- 3 : lorsque deux cercles incluent le centre de l'autre, ils sont en double inclusion :
- 1 : lorsque seulement un de deux cercles s'intersectent sans y inclure leurs centres, ils sont en intersection :



- 2 : lorsque seulement un de deux cercles inclue le centre de l'autre, ils sont en simple inclusion :
- 0 : lorsque deux cercles ne s'intersectent pas, ils sont en exclusion :



Le prototype de la fonction sera le suivant :

```
char inclusion_cercles (int c1x, int c1y, int c1r, int c2x, int c2y, int c2r) ;
```

b) Pour tester la fonction du (a), écrire un **PROGRAMME** en **LANGAGE C** qui vérifiera si la communication est possible entre deux objets de positions $(-2; 3)$ et $(2; 3)$ pour lesquels l'utilisateur indiquera le rayon de leurs portées respectives. Le programme indiquera alors si une communication est possible (bi-directionnelle, cas 3, ou uni-directionnelle, cas 2) ou impossible (cas 1 ou cas 0).

Ci-après quatre exemples d'exécution de ce programme :

<pre> Invite de commandes C:\ExemplesC> exercice1b.exe Donnez le premier rayon : 1 Donnez le deuxieme rayon : 1 Les objets ne peuvent pas communiquer. C:\ExemplesC> </pre>	<pre> Invite de commandes C:\ExemplesC> exercice1b.exe Donnez le premier rayon : 2 Donnez le deuxieme rayon : 5 Les objets peuvent communiquer. C:\ExemplesC> </pre>
<pre> Invite de commandes C:\ExemplesC> exercice1b.exe Donnez le premier rayon : 3 Donnez le deuxieme rayon : 3 Les objets ne peuvent pas communiquer. C:\ExemplesC> </pre>	<pre> Invite de commandes C:\ExemplesC> exercice1b.exe Donnez le premier rayon : 5 Donnez le deuxieme rayon : 5 Les objets peuvent communiquer. C:\ExemplesC> </pre>

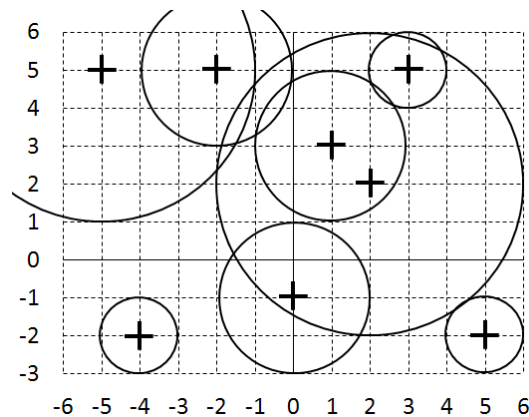
c) Pour amorcer le développement du logiciel de calcul, écrire un **PROGRAMME** en **LANGAGE C** dans lequel sont déclarés les trois tableaux suivants, définissant 8 objets communicants et le rayon de leur portée :

```

int bx[8] = { -2, 3, -4, 5, -5, 1, 0, 2 } ;
int by[8] = { 5, 5, -2, -2, 5, 3, -1, 2 } ;
int br[8] = { 2, 1, 1, 1, 4, 2, 2, 4 } ;

```

Par exemple, le premier objet communicant est positionné en $(-2; 5)$ et le rayon de sa portée est de 2. Ces objets communicants et leurs portées ont été représentés graphiquement sur la figure suivante :



Le programme commencera par demander à l'utilisateur de saisir la position et le rayon de la portée d'un nouvel objet communicant, puis affichera (à l'aide d'une boucle) le nombre d'objets avec lesquels il peut communiquer de manière bidirectionnelle ou uni-directionnelle. Ce programme doit lui aussi utiliser la fonction du (a).

Ci-après trois exemples d'exécution de ce programme :

<pre> Invite de commandes C:\ExemplesC> exercicelc.exe Donnez l'abscisse : -4 Donnez l'ordonnee : 0 Donnez le rayon : 1 Cet objet peut communiquer : en bi-directionnel avec 0 objets. en uni-directionnel avec avec 0 objets. C:\ExemplesC> </pre>	<pre> Invite de commandes C:\ExemplesC> exercicelc.exe Donnez l'abscisse : -3 Donnez l'ordonnee : 5 Donnez le rayon : 3 Cet objet peut communiquer : en bi-directionnel avec 2 objets. en uni-directionnel avec avec 0 objets. C:\ExemplesC> </pre>
<pre> Invite de commandes C:\ExemplesC> exercicelc.exe Donnez l'abscisse : -4 Donnez l'ordonnee : 0 Donnez le rayon : 3 Cet objet peut communiquer : en bi-directionnel avec 0 objets. en uni-directionnel avec avec 1 objets. C:\ExemplesC> </pre>	<pre> Invite de commandes C:\ExemplesC> exercicelc.exe Donnez l'abscisse : 2 Donnez l'ordonnee : 4 Donnez le rayon : 2 Cet objet peut communiquer : en bi-directionnel avec 1 objets. en uni-directionnel avec avec 2 objets. C:\ExemplesC> </pre>

E Être capable d'écrire des fonctions et des programmes simples

Exercice 2 : Paquetage d'objets communicants

Nous souhaitons créer un paquetage permettant de manipuler des objets communicants, décrits chacun par sa position et le rayon de la portée de son émetteur. Une partie de ce paquetage est déjà écrite, mais il manque encore deux fonctions. L'objectif de l'exercice est d'écrire les deux fonctions manquantes.

Ci-dessous le fichier d'entête `TObjet.h` déjà écrit :

```

1 /* File: TObjet.h */
2 #ifndef TOBJET_H
3 #define TOBJET_H
4
5 struct objet {
6     int x, y, r ;
7 } ;
8
9 typedef struct objet TObjet ;
10
11 void objet_print (TObjet *b) ;
12
13 char objet_read (TObjet *b, FILE *desc) ;
14
15 char objet_communicate (TObjet *b1, TObjet *b2) ;
16
17 #endif /* TOBJET_H */

```

Ci-dessous le fichier d'implantation `TObjet.c` partiellement écrit :

```

1 /* File: TObjet.c */
2 #include <stdio.h>
3 #include <math.h>
4
5 #include "TObjet.h"
6
7 char inclusion_cercles (int c1x, int c1y, int c1r, int c2x, int c2y, int c2r) ;
8
9 void objet_print (TObjet *b) {
10     printf ("(%2d;%2d)...\d\n", b->x, b->y, b->r) ;
11 }

```

a) La première **FONCTION** en **LANGAGE C** manquante permet de lire un (et un seul) objet depuis un descripteur de fichier ouvert en lecture. Nous considérons que le descripteur de fichier est déjà correctement créé et que chaque ligne du fichier ouvert est structurée comme suit :

- La première ligne indique le nombre d'objets contenus dans le fichier ;
- Chacune des lignes suivantes décrit un et un seul objet ;
- Chacune de ces lignes donne dans l'ordre l'abscisse et l'ordonnée de la position de l'objet, puis le rayon de la portée de son émetteur.

Ci-contre l'exemple du fichier `detection.txt` contenant huit objets communicants (correspondant à la figure de l'exercice 1).

Écrivez le code de cette fonction `objet_read()` permettant d'initialiser un objet en lisant depuis un descripteur ouvert vers un tel fichier.

```

1 8
2 -2 5 2
3 3 5 1
4 -4 -2 1
5 -5 -2 1
6 -5 5 4
7 1 3 2
8 0 -1 2
9 2 2 4

```

b) La deuxième **FONCTION** en **LANGAGE C** manquante permet de savoir si deux objets peuvent communiquer. Pour ce faire, elle doit tout simplement réutiliser et renvoyer la valeur de la fonction `inclusion_cercles()` de l'exercice 1-a. Écrivez le code de cette fonction `objet_communicate()`.

Remarque : Les fonctions utiles pour la manipulation de fichiers sont décrites en annexe 1 (page 6).

D Être capable de lire dans des fichiers texte et de manipuler une structure de données

Exercice 3 : Programme d'analyse des communications entre objets communicants

Écrire un **PROGRAMME** en **LANGAGE C** qui demande de spécifier le nom d'un fichier contenant la liste des objets communicants qui ont été détectés, puis qui lit et stocke les objets communicants contenus dans ce fichier. Ensuite, le programme affiche quels objets peuvent communiquer entre eux, en détaillant si la communication est bidirectionnelle ou uni-directionnelle. Enfin, le programme indique les objets isolés qui sont dans l'impossibilité de communiquer avec les autres objets. Ce programme utilisera habilement les structures de données et les fonctions du paquetage de l'exercice 2.

Ci-dessous un exemple d'exécution de ce programme :

```

C:\ExemplesC> exercice3.exe
Donnez le nom du fichier de previsions : detection.txt
Ces deux objets peuvent communiquer en uni-directionnel :
(-2 ; 5) ... 2
(-5 ; 5) ... 4
Ces deux objets peuvent communiquer en uni-directionnel :
( 3 ; 5) ... 1
( 2 ; 2) ... 4
Ces deux objets peuvent communiquer en bi-directionnel :
( 1 ; 3) ... 2
( 2 ; 2) ... 4
Ces deux objets peuvent communiquer en uni-directionnel :
( 0 ; -1) ... 2
( 2 ; 2) ... 4
2 objets sont hors atteinte :
(-4 ; -2) ... 1
(-5 ; -2) ... 1
C:\ExemplesC>

```

Les deux macro-fonctions suivantes pourront être utilisées afin de faciliter la programmation :

```

7 #define FOPEN(_d,_f,_m) {\
8   if (!(_d = fopen (_f, _m))) {\
9     perror("Error: cannot open file.");\
10    exit (1) ;\
11   }}
12
13 #define MALLOC(_p,_t,_n) {\
14   if (!(_p = malloc (_n * sizeof (_t)))) {\
15     perror("Error: memory allocation failed.");\
16     exit (1) ;\
17   }}

```

- Un fichier peut ainsi être ouvert comme suit : FOPEN (descripteur, nom, mode)
Par exemple, pour ouvrir le fichier `scores.txt` en écriture :
`FILE *desc ;`
`FOPEN (desc, "scores.txt", "w") ;`
- Un espace mémoire peut ainsi être alloué comme suit : MALLOC (pointeur, type, taille)
Par exemple, pour allouer dynamiquement un tableau de dix entiers :
`int *scores ;`
`MALLOC (scores, int, 10) ;`

Aide : deux boucles imbriquées sont nécessaires.

Remarque : Les fonctions utiles pour la manipulation de fichiers sont décrites en annexe 1 (page 6).

C Être capable d'utiliser des pointeurs et des structures de données

Exercice 4 : Algorithmique et Structures de Données

Annexe 1 : Extraits de la section 3 des pages de manuel Linux

1. NOM

fopen - Fonctions d'ouverture de flux

2. SYNOPSIS

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

```
FILE *fopen(char *path, char *mode);
```

3. DESCRIPTION

La fonction **fopen()** ouvre le fichier dont le nom est contenu dans la chaîne pointée par **path** et lui associe un flux.

L'argument **mode** pointe vers une chaîne commençant par l'une des séquences suivantes (éventuellement suivie par des caractères supplémentaires, conformément à la description ci-dessous).

r : Ouvre le fichier en lecture. Le pointeur de flux est placé au début du fichier.

w : Tronque le fichier à son début ou ouvre le fichier en écriture. Le pointeur de flux est placé au début du fichier.

a : Ouvre le fichier en ajout (écriture à la fin du fichier). Le fichier est créé s'il n'existait pas. Le pointeur de flux est placé à la fin du fichier.

4. VALEUR RENVOYÉE

Si elle réussit intégralement, **fopen()** renvoie un pointeur de type **FILE**. Sinon, elle renvoie **NULL** et **errno** contient le code d'erreur.

1. NOM

perror - Afficher un message d'erreur système

2. SYNOPSIS

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

```
void perror(char *s);
```

```
#include <errno.h>
```

```
int errno;
```

3. DESCRIPTION

La fonction **perror()** affiche un message sur la sortie d'erreur standard, décrivant la dernière erreur rencontrée durant un appel système ou une fonction de bibliothèque décrite par **errno**.

Notez que **errno** est indéfinie après un appel de fonction de bibliothèque réussi. Cette fonction peut modifier **errno** même si elle réussit, ne serait-ce que par des appels système internes qui peuvent échouer. Ainsi, si un appel qui échoue n'est pas immédiatement suivi par **perror()**, la valeur de **errno** doit être sauvegardée.

1. NOM

feof - Vérifier l'état d'un flux

2. SYNOPSIS

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

```
int feof(FILE *stream);
```

3. DESCRIPTION

La fonction **feof()** teste l'indicateur de fin de fichier du flux pointé par **stream** et renvoie une valeur non nulle si cet indicateur est actif.

1. NOM

fscanf - Entrées formatées

2. SYNOPSIS

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

```
int fscanf(FILE *stream, char *format, ...);
```

3. DESCRIPTION

La fonction **fscanf()** lit ses entrées depuis le flux pointé par **stream**. La fonction **fscanf()** analyse les entrées au même format que la fonction **scanf()**.

4. VALEUR RENVOYÉE

La fonction **fscanf()** renvoie le nombre d'éléments d'entrées correctement mis en correspondance et assignés. Ce nombre peut être plus petit que le nombre d'éléments attendus, et même être nul, s'il y a une erreur de mise en correspondance. Le filtre pour lire une ligne entière sous forme d'une chaîne de caractères est `%[^\r\n]s`.

1. NOM

fprintf - Formatage des sorties

2. SYNOPSIS

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

```
int fprintf(FILE *stream, char *format, ...);
```

3. DESCRIPTION

La fonction **fprintf()** écrit sa sortie sur le flux **stream** indiqué. Comme la fonction **printf()**, cette fonction crée sa sortie sous le contrôle d'une chaîne de format qui indique les conversions à apporter aux arguments qui suivent.

3.1. VALEUR RENVOYÉE

En cas de succès, la fonction **fprintf()** renvoie le nombre de caractères affichés (sans compter l'octet nul final utilisé pour terminer les sorties dans les chaînes).

1. NOM

fclose - Fermer un flux

2. SYNOPSIS

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

```
int fclose(FILE *fp);
```

3. DESCRIPTION

La fonction **fclose()** vide le flux pointé par **fp** (en écrivant toute donnée de sortie en tampon avec **fflush(3)**) et ferme le descripteur de fichier sous-jacent.

4. VALEUR RENVOYÉE

Si la fonction réussit intégralement, elle renvoie 0, sinon elle renvoie EOF et **errno** contient le code d'erreur. Dans tous les cas, tout autre accès ultérieur au flux (y compris un autre appel de **fclose()**) conduit à un comportement indéfini.