Examen de Programmation C

ING1 – GI

12 janvier 2022

— *Durée : 2 heures*

— *Vous devez rédiger votre copie à l'aide d'un* ***stylo à encre*** *exclusivement.*

— *Toutes vos affaires (sacs, vestes, trousse, etc.) doivent être placées à l'avant de la salle.*

— *Aucun document n'est autorisé.*

— *Aucune machine électronique ne doit se trouver sur vous ou à proximité, même éteinte.*

— *Aucun déplacement n'est autorisé.*

— ***Aucune question au professeur n'est autorisée****. Si vous pensez avoir détecté une erreur, continuez en expliquant les hypothèses que vous faites.*

— *Aucun échange, de quelque nature que ce soit, n'est possible.*

— *Le barème est donné à titre indicatif.*

Exercice 1 : Cours et application directe (8 pts)

1. À quoi sert l'instruction `#ifndef ?` (0.5 pt)
2. Que fait la fonction `free`? (0.5 pt)
3. Pourquoi certaines cibles du Makefile sont définies en `PHONY`? (1pt)
4. Bob a un dossier qui contient les sources *toto.c*, *toto.h*, *tata.c*, *tata.h*, *titi.c* et *titi.h*. Bob écrit le Makefile suivant, mais une erreur s'y est glissé. À quelle ligne se situe l'erreur? Comment la corriger? (1pt)

Listing 1 – Makefile de Bob

```
1 #le symbole ___ indique une tabulation
2 all : prog
3
4 prog : toto.o tata.o titi.o
5 ___gcc $^ -o prog
6
7 .c.o :
8 ___gcc $^ -o $@
```

5. Quelle est l'utilité des paramètres de la fonction `int main (int argc, char** argv) ?` (1pt)
6. Quelles sont les différences (avantages et inconvéniants) entre une bibliothèque statique et une bibliothèque dynamique? (1pt)

7. Qu'affiche le programme suivant ? (1pt)

Listing 2 – Échange de valeurs

```c
void echange (int a, int* b) {
    int tmp;
    tmp = a;
    a = *b;
    *b = tmp;
}
int main (int argc, char** argv) {
    int a;
    int b;
    a = 4;
    b = 2;
    echange(a, &b);
    printf("a = %d \n b = %d \n", a, b);
    return (0);
}
```

8. Pour les 4 programmes suivant : s'ils sont corrects, donner la valeur de *a* et de ce qui est pointé par *b*. Sinon, expliquer les erreurs. (2pts)

(a) Programme 1

```c
int main (int argc, char** argv) {
    int a;
    int* b;
    a = 4;
    *b = a+1;
    return (0);
}
```

(b) Programme 2

```c
int main (int argc, char** argv) {
    int a;
    int* b;
    a = 4;
    &b = a;
    return (0);
}
```

(c) Programme 3

```c
int main (int argc, char** argv) {
    int a;
    int* b;
    a = 4;
    b = &a;
    return (0);
}
```

(d) Programme 4

```c
int main (int argc, char** argv) {
    int a;
    int* b;
    a = 4;
    b = (&a)+1;
    return (0);
}
```

Exercice 2 : Jouer sur les mots (3 pts)

1. Écrire la procédure `void remplacer(char* mot, char c1, char c2)` qui remplace le caractère `c1` par celui de `c2` dans le `mot`, **sans utiliser** les fonctions de la librairie *string.h*. (1pt)
2. Écrire la procédure `char* prefixe(char* mot, char c1)` qui retourne la sous-chaîne de `mot` allant du début de la chaîne jusqu'au premier caractère `c1` (non inclus dans le préfixe). (2pts)
   Exemple : `prefixe("bonjour",'j')` renvoie `"bon"`

Exercice 3 : Matrice tridiagonale (3 pts)

Un tableau bidimensionnel est généralement utilisé pour stocker une matrice. Chaque entrée du tableau représente un élément $a_{i,j}$ de la matrice, ce qui nécessite alors au moins $(n \times m)$ espaces mémoire pour représenter une matrice de dimension $n \times m$. Une matrice tridiagonale est une matrice dont tous les coefficients qui ne sont ni sur la diagonale principale, ni sur la diagonale juste au-dessus, ni sur la diagonale juste en dessous, sont nuls. Afin d'optimiser la manière de stocker les matrices tridiagonale, on applique la transformation suivante :

$$A_{n,n} = \begin{bmatrix} a_1 & b_1 & 0 & \cdots & \cdots & 0 \\ c_2 & a_2 & b_2 & \ddots & & \vdots \\ 0 & c_3 & a_3 & b_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & b_{n-1} \\ 0 & \cdots & \cdots & 0 & c_n & a_n \end{bmatrix} \quad \Rightarrow \quad B_{3,n} = \begin{bmatrix} 0 & c_2 & c_3 & \cdots & c_{n-1} & c_n \\ a_1 & a_2 & a_3 & \cdots & a_{n-1} & a_n \\ b_1 & b_2 & b_3 & \cdots & b_{n-1} & 0 \end{bmatrix}$$

1. Écrire la fonction `double** transformer(double** mat, int n)` qui retourne la matrice `mat` optimisée en espace en suivant la transformation décrite ci-dessus. (3pts)

Exercice 4 : Frise chronologique (6 pts)

On souhaite placer des événements sur une frise chronologique. Pour cela, nous avons besoin de stocker les différents événements dans le bon ordre et dans un fichier.

1. Un événement est défini par une date (jour, mois, année) et d'un descriptif.
   Définir la structure `Evenement`. (1pt)
2. Écrire la fonction `compare` qui permet de comparer deux événements. Les deux événements sont passés en paramètre. Elle retourne 0 si les deux événements sont à la même date, -1 si le premier événement est avant le deuxième, et 1 sinon. On se basera sur l'ordre chronologique. (1pt)

3. Écrire la procédure `triEvenements` qui permet de trier un tableau d'`Evenement` passé en paramètre. (2pts)
4. Écrire la procédure `sauvegarder` qui permet d'écrire dans le fichier "frise.txt" un tableau d'`Evenement` passé en paramètre. (2pts)

Rappel :
Le tri à bulles consiste à comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils ne sont pas dans l'ordre. Après un premier parcours complet du tableau, le plus grand élément est forcément en fin de tableau, à sa position définitive. Le reste du tableau est cependant encore en désordre. Il faut donc répéter les parcours du tableau, jusqu'à ce que tous les éléments soient placés à leur position définitive.
Principe :

— On effectue une succession de parcours du tableau.

— À chaque parcours, on compare tout couple de cases consécutives $A[j]$ et $A[j+1]$. Si ces deux cases ne sont pas ordonnées, alors on les échange.

— Lorsqu'aucun échange n'est effectué lors d'un parcours, alors le tableau est trié et l'algorithme s'arrête.

A Annexes

A.1 Extrait de la page de man : strtok

SYNOPSIS

```
#include <string.h>
char *strtok(char *restrict str, const char *restrict delim);
```

DESCRIPTION

The **strtok()** function breaks a string into a sequence of zero or more nonempty tokens. On the first call to **strtok()**, the string to be parsed should be specified in *str*. In each subsequent call that should parse the same string, *str* must be NULL.
The *delim* argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in *delim* in successive calls that parse the same string.
Each call to **strtok()** returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, **strtok()** returns NULL.
A sequence of calls to **strtok()** that operate on the same string maintains a pointer that determines the point from which to start searching for the next token. The first call to **strtok()** sets this pointer to point to the first byte of the string. The start of the next token is determined by scanning forward for the next nondelimiter byte in *str*. If such a byte is found, it is taken as the start of the next token. If no such byte is found, then there are no more tokens, and **strtok()** returns NULL. (A string that is empty or that contains only delimiters will thus cause **strtok()** to return NULL

on the first call.)
The end of each token is found by scanning forward until either the next delimiter byte is found or until the terminating null byte ('\0') is encountered. If a delimiter byte is found, it is overwritten with a null byte to terminate the current token, and **strtok()** saves a pointer to the following byte; that pointer will be used as the starting point when searching for the next token. In this case, **strtok()** returns a pointer to the start of the found token.

RETURN VALUE

The **strtok()** functions return a pointer to the next token, or NULL if there are no more tokens.

A.2 Extrait de la page de man : strdup

SYNOPSIS

```
#include <string.h>
char *strdup(const char *s);
```

DESCRIPTION

The **strdup()** function returns a pointer to a new string which is a duplicate of the string *s*. Memory for the new string is obtained with **malloc(3)**, and can be freed with **free(3)**.

RETURN VALUE

On success, the **strdup()** function returns a pointer to the duplicated string. It returns NULL if insufficient memory was available, with *errno* set to indicate the cause of the error.

A.3 Extrait de la page de man : strstr

SYNOPSIS

```
#include <string.h>
char *strstr(const char *haystack, const char *needle);
```

DESCRIPTION

The **strstr()** function finds the first occurrence of the substring *needle* in the string *haystack*. The terminating null bytes ('\0') are not compared.

RETURN VALUE

This function return a pointer to the beginning of the located substring, or NULL if the substring is not found.

A.4 Extrait de la page de man : strcpy, strncpy

SYNOPSIS

```
#include <string.h>
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

DESCRIPTION

The **strcpy()** function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!*
The **strncpy()** function is similar, except that at most *n* bytes of *src* are copied. **Warning**: If there is no null byte among the first *n* bytes of *src*, the string placed in *dest* will not be null-terminated.
If the length of *src* is less than *n*, **strncpy()** writes additional null bytes to *dest* to ensure that a total of *n* bytes are written.

RETURN VALUE

The **strcpy()** and **strncpy()** functions return a pointer to the destination string *dest*.

A.5 Extrait de la page de man : fopen

SYNOPSIS

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char *mode);
```

DESCRIPTION

The **fopen()** function opens the file whose name is the string pointed to by *pathname* and associates a stream with it.
The argument *mode* points to a string beginning with one of the following sequences:

r Open text file for reading. The stream is positioned at the beginning of the file.

r+ Open for reading and writing. The stream is positioned at the beginning of the file.

w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

a Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

a+ Open for reading and appending (writing at end of file). The file is created if it does not exist. Output is always appended to the end of the file.

RETURN VALUE

Upon successful completion **fopen()** return a FILE pointer. Otherwise, NULL is returned and *errno* is set to indicate the error.

A.6 Extrait de la page de man : fclose

SYNOPSIS

```
#include <stdio.h>
int fclose(FILE *stream);
```

DESCRIPTION

The **fclose()** function flushes the stream pointed to by *stream* (writing any buffered output data using **fflush(3)**) and closes the underlying file descriptor.
The behaviour of **fclose()** is undefined if the *stream* parameter is an illegal pointer, or is a descriptor already passed to a previous invocation of **fclose()**.

RETURN VALUE

Upon successful completion, 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error. In either case, any further access (including another call to **fclose()**) to the *stream* results in undefined behavior.

A.7 Extrait de la page de man : printf, fprintf

SYNOPSIS

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

DESCRIPTION

The functions in the **printf()** family produce output according to a *format* as described below. The function **printf()** write output to *stdout*, the standard output stream; **fprintf()** write output to the given output *stream*.
All of these functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of **stdarg(3)**) are converted for output.

RETURN VALUE

Upon successful return, these functions return the number of characters printed (excluding the null byte used to end output to strings).
If an output error is encountered, a negative value is returned.