



---

# Programmation Avancée en C

---

Auteur(s) : Florian Legendre



# Légendes et Abbréviations utilisées

**Question :** Ceci est une question de l'enseignant<sup>1</sup>

**Réponse :** Ceci est une réponse de l'enseignant ou validée par l'enseignant

**Réponse :** *Ceci est une réponse du ou d'un des auteurs non validée par l'enseignant*

- 1 Ceci est du code source.
- 2 Selon les langages, différents mots seront colorés selon si ce sont des mots clefs ou non (comme `int`, `char`, etc.).

**Listing 1** – Exemple de code source

Ceci est un formatage automatique Latex d'un texte copié–collé directement depuis un terminal Bash ayant valeur de capture d'écran. La coloration correspond à une coloration quelconque d'un terminal Bash (les chemins étant habituellement coloré et le nom de l'utilisateur aussi comme `crex@crex:~$ ...`)

**Listing 2** – Exemple d'une pseudo capture d'écran Bash

Ceci est un formatage automatique Latex d'un texte copié–collé directement depuis un terminal Bash et dont la coloration a été revisitée pour mettre en relief les commandes natives de Bash comme `cp` en cyan, les alias ou scripts faits par l'utilisateur en jaune (Ex. `mygcc`), les options en magenta (comme `-i`), les opérateurs de redirection en rouge (comme `>` | `<`) et l'invite de commande `$` en vert.

**Listing 3** – Exemple d'une capture formaté en style 'Tutoriel de Commande Bash' ou TCB

---

1. Je tiens à remercier Gilles Subrenat qui m'a autorisé à reproduire dans ce document une partie de ses supports d'enseignement

# Table des matières

<b>I Synthèse</b>	<b>6</b>
<b>1 Rappels Bash et Allocation Dynamique</b>	<b>7</b>
1.1 Bash . . . . .	7
1.1.1 Compiler avec gcc sous Bash . . . . .	7
1.1.2 Créer des alias . . . . .	7
1.1.3 Gérer des archives . . . . .	8
1.2 Allocation Dynamique . . . . .	8
<b>2 Les chaînes de caractère en C</b>	<b>9</b>
2.1 Propriétés des chaînes de caractère en C . . . . .	9
2.1.1 Le Null-Byte . . . . .	9
2.1.2 Mutabilité, non mutabilité de la chaîne . . . . .	9
2.2 Fonctions de manipulations des chaînes . . . . .	10
2.2.1 Manipulations . . . . .	10
2.2.2 Conversions . . . . .	10
2.3 Taille de buffers . . . . .	10
<b>3 Modules</b>	<b>11</b>
3.1 Notions importantes pour la conception . . . . .	11
3.1.1 Déclarer vs Définir . . . . .	11
3.1.2 .c vs .h . . . . .	11
3.1.3 Protéger des inclusions multiples . . . . .	11
3.1.4 Comment éviter les inclusions croisées ? . . . . .	12
3.1.5 Initialisations vs Affectations . . . . .	12
3.2 Le mot-clef static . . . . .	12
3.3 Le mot-clef extern . . . . .	12
<b>4 Fonctions de gesticions de fichiers en C</b>	<b>13</b>
4.1 Prog Système et bonnes pratiques pour rester zen . . . . .	13
4.2 Les fonctions de manipulations de fichiers . . . . .	13
4.2.1 Haut-niveau (bufferisé) . . . . .	13
4.2.2 Bas-niveau (non bufferisé) . . . . .	13
4.3 Exemple de module pour la manipulation de fichiers . . . . .	14
4.4 Ecriture de structures dans un fichier et lecture . . . . .	15
<b>5 Abstraction pointeur</b>	<b>16</b>
5.1 Principe . . . . .	16
5.2 Exemple de mise en oeuvre . . . . .	16
<b>6 Tubes, Forks et Exec</b>	<b>17</b>
6.1 Forks/Execs . . . . .	17
6.1.1 D'abord le fork()... . . . . .	17
6.1.2 Puis le Exec! . . . . .	17
6.2 Tubes . . . . .	18
6.2.1 Principe . . . . .	18

6.2.2	Tubes nommés . . . . .	19
6.2.3	Tubes anonymes . . . . .	19
6.2.4	Tubes vs Fork/Exec . . . . .	21
<b>7</b>	<b>Sémaphores</b>	<b>23</b>
7.1	Principe . . . . .	23
7.2	Exemple de module pour gérer les sémaphores . . . . .	23
<b>8</b>	<b>Threads</b>	<b>27</b>
8.1	Pointeurs sur fonction . . . . .	27
8.2	Manipulation les Threads . . . . .	27
8.3	Un exemple de module . . . . .	28
8.4	Passer des arguments aux threads . . . . .	28
8.5	Gérer la concurrence mémoire entre threads . . . . .	28
8.6	Exemple d'algorithme pour l'utilisation des threads . . . . .	29
<b>II</b>	<b>Analyse</b>	<b>31</b>
<b>9</b>	<b>Rappels Bash et Allocation Dynamique</b>	<b>32</b>
9.1	Résumé et cadre de travail . . . . .	32
9.2	Bash . . . . .	32
9.3	Exercice 3 : Sempiternel Hello World . . . . .	33
9.4	Exercice 4 : Archive, Compression, Somme de contrôle . . . . .	34
9.5	Exercice 5 : Redirections . . . . .	37
9.5.1	Un programme pour manipuler des flux d'entrées et de sorties . . . . .	37
9.5.2	Manipulation sur les opérateurs de redirections > et   . . . . .	37
9.5.3	Quelques exercices d'anticipation sur les résultats de redirections . . . . .	40
9.5.4	Rediriger une sortie erreur dans un tube ? . . . . .	41
9.6	Exercice 6 : Variables Shell et Programmes C . . . . .	42
9.7	Exercice 7 : Variables Shell et Programmes C (suite) . . . . .	44
9.8	Exercice 8 : Relais . . . . .	46
9.9	Exercice 9 : B.a.-ba . . . . .	47
9.10	Exercice 10 : Utilisation des Antiquotes . . . . .	48
9.11	Exercice 11 : Pointeurs et Allocation Dynamique . . . . .	50
9.11.1	Pointeurs sans allocation . . . . .	50
9.11.2	Oubli d'allocation . . . . .	50
9.11.3	Non-sens . . . . .	54
9.11.4	Allocation de structures . . . . .	54
9.11.5	Tableau à deux dimensions . . . . .	57
9.11.6	Affectation de tableaux ? . . . . .	62
9.12	Exercice 12 : Tableaux . . . . .	63
<b>10</b>	<b>Les chaînes de caractère en C</b>	<b>66</b>
10.1	Exercice 1 : Manipulations Standard . . . . .	66
10.1.1	Exercice 1a : Différence entre caractère, chaîne de caractères mutable ou non . . . . .	66
10.1.2	Exercice 1b : Affectation de chaînes . . . . .	71
10.1.3	Exercice 1c : Palindrome ? . . . . .	72
10.1.4	Exercice 1d : Retournement d'une chaîne . . . . .	74
10.1.5	Exercice 1e : Version personnelle de strlen . . . . .	75
10.1.6	Exercice 1f : Version personnelle de strcpy . . . . .	76
10.1.7	Exercice 1g : strncpy (utilisation) . . . . .	77
10.1.8	Exercice 1h : strcmp (utilisation) . . . . .	78
10.1.9	Exercice 1i : strcmp (version personnelle) . . . . .	79
10.1.10	Exercice 1j : streat (utilisation) . . . . .	80
10.1.11	Exercice 1k : strstr (utilisation) . . . . .	81

10.1.12 Exercice 11 : sprintf ou commenter formater une chaîne . . . . .	83
10.1.13 Exercice 1m : Du bon usage des fonctions . . . . .	85
10.1.14 Exercice 1n : == ou strcmp ? . . . . .	86
10.2 Exercice 2 : Allocation Dynamique . . . . .	88
10.2.1 Exercice 2a : Clone d'une chaîne . . . . .	88
10.2.2 Exercice 2b : strcat version dynamique . . . . .	89
10.2.3 Exercice 2c : Formatage et allocation . . . . .	90
10.2.4 Exercice 2d : Retour sur les chaînes littérales . . . . .	91
<b>11 Modules</b>	<b>92</b>
11.1 Définir ou déclarer ? . . . . .	92
11.1.1 Explications . . . . .	92
11.1.2 Exercice . . . . .	93
11.1.3 Exercice . . . . .	94
11.2 Notion de module . . . . .	95
11.2.1 Définitions . . . . .	95
11.2.2 Exercice . . . . .	96
11.3 Mot-clé static . . . . .	99
11.3.1 Fonctions static . . . . .	99
11.3.2 Variable static définie hors des fonctions . . . . .	101
11.3.3 Variable static locale à une fonction . . . . .	102
11.4 Mot-clé extern . . . . .	104
11.5 Fichiers d'entête .h . . . . .	105
11.5.1 Définition . . . . .	105
11.5.2 Problématique des inclusions multiples . . . . .	105
11.5.3 Solution non retenue . . . . .	105
11.5.4 Solution retenue . . . . .	106
11.5.5 Exercice . . . . .	107
11.5.6 Inclusions croisées . . . . .	111
11.5.7 Exercice . . . . .	112
11.6 Initialisation vs. affectation . . . . .	113
11.6.1 Explications . . . . .	113
11.6.2 Exercice . . . . .	114
11.6.3 Exercice . . . . .	116
11.7 Exercice complet . . . . .	118
11.7.1 Module “mathématiques financières” . . . . .	118
11.7.2 Module “gestion de comptes” . . . . .	120
11.7.3 Module du programme principal : main . . . . .	122
11.7.4 Compilation . . . . .	124
<b>12 Fonctions de gestions de fichiers en C</b>	<b>125</b>
12.1 Résumé et cadre de travail . . . . .	125
12.2 Mode bufferisé ou non . . . . .	125
12.3 Mode binaire ou mode texte . . . . .	126
12.4 Liste de quelques fonctions . . . . .	127
12.5 Exercices . . . . .	128
12.5.1 Exercice A . . . . .	128
12.5.2 Exercice B : curiosité binaire . . . . .	134
12.5.3 Exercice C : écriture/lecture d'une chaîne de caractères . . . . .	135
12.5.4 Exercice D : string input/output (bis repetita) . . . . .	137
12.5.5 Exercice E : tableaux d'entiers en binaire . . . . .	139
12.5.6 Exercice F : tableaux d'entiers en mode texte . . . . .	141
12.5.7 Exercice G : Lecture / écriture d'une structure . . . . .	144
12.5.8 Exercice H . . . . .	146
12.5.9 Exercice I . . . . .	147

<b>13 Abstraction pointeur</b>	<b>148</b>
13.1 Exercice 1 . . . . .	148
13.2 Exercice 2 . . . . .	152
13.3 Exercice 3 . . . . .	153
13.4 Exercice 4 . . . . .	163
13.4.1 Implémentation de Voiture.c . . . . .	163
13.4.2 Intégration de Collection.c de l'exercice et Voiture.c de l'exercice 4a . . . . .	169
<b>14 Tubes, Forks et Exec</b>	<b>170</b>
14.1 Résumé et cadre de travail . . . . .	170
14.2 Principe de fonctionnement des tubes . . . . .	171
14.3 Bash et tubes nommés . . . . .	172
14.4 Programmation et tubes nommés . . . . .	174
14.5 Programmation et tubes anonymes . . . . .	178
14.6 Programmation et tubes anonymes : bidirectionnel . . . . .	180
14.7 Exec . . . . .	182
14.8 Fork, exec et tubes anonymes . . . . .	186
14.9 Fork, exec et tubes nommés . . . . .	190
14.9.1 Chef d'orchestre . . . . .	190
14.9.2 Worker/master 1 . . . . .	193
14.9.3 Worker/master 2 . . . . .	195
14.9.4 Worker/master 3 . . . . .	196
<b>15 Sémaphores</b>	<b>197</b>
15.1 Résumé et Cadre de travail . . . . .	197
15.2 Principe de fonctionnement des sémaphores IPC . . . . .	197
15.3 Création et destruction de sémaphores . . . . .	198
15.4 Utilisation basique d'un sémaphore . . . . .	200
15.5 Sémaphore et processus indépendants . . . . .	203
15.6 Sémaphore et processus indépendants (version 2) . . . . .	206
15.7 Sémaphore et barrière de synchronisation . . . . .	209
15.8 Mémoire partagée . . . . .	213
15.9 File de messages . . . . .	219
15.10 Paquetage de sémaphore . . . . .	226
<b>16 Threads</b>	<b>235</b>
16.1 Résumé et cadre de travail . . . . .	235
16.2 Principe de fonctionnement des threads . . . . .	236
16.3 Création de threads . . . . .	237
16.4 Création de threads avec paramètres . . . . .	239
16.5 Création de threads avec paramètres (bis repetita) . . . . .	243
16.6 Mutex posix . . . . .	246
16.7 Sémaphore anonyme ( facultatif ) . . . . .	254

Première partie

Synthèse

# Chapitre 1

## Rappels Bash et Allocation Dynamique

### Bash

#### 1.1.1 Compiler avec gcc sous Bash

gcc est composé de trois grandes fonctionnalités :

- Le préprocesseur (remplacer les macros définies dans les `#define`, faire les `#include`, bref tout ce qui commence par un `#<blabla>`. Ceci est par ailleurs appelé "directive préprocesseur" en C)
- Le compilateur (crée des .o, convertit en langage machine)
- L'éditeur de lien (lie les .o entre eux selon les include donnés).

Pour appeler le préprocesseur : `gcc E nom_fichier`

Pour appeler le compilateur : `gcc -c fichier.c → fichier.o`

Pour appeler l'éditeur de lien : `gcc -o nom_executable fichier.o → nom_executable`

gcc prend un certain nombre de drapeaux (flags) sous forme d'options (sous Bash) qui peuvent modifier son comportement... Ce que gcc autorise ou non à la compilation par exemple :

- `-Wall` → Active de nombreux warnings utiles pour les débutants en C
- `-Wextra` → Active davantage de warnings utiles pour les débutants en C
- `-Werror` → Tous les warnings sont considérés comme des erreurs
- `-pedantic` → Active un warning si le standard indiqué n'est pas utilisé
- `-std=c99` → Détermine le standard utilisé
- `-g` → Permet de laisser des informations au débuggeur GDB

#### 1.1.2 Créer des alias

Souvent pour éviter de devoir sans cesse retaper une commande avec des options qu'on utilise souvent (et/ou pour éviter de retaper sans cesse une suite de commandes) on définit un alias. Cet alias peut être défini dans `~/.bash_aliases` ou, s'il n'existe pas, dans `~/.bash_rc` :

- `alias mygcc='gcc -Wall -Wextra -Werror -pedantic -std=c99 -g'`
- `alias softgcc='gcc -Wall -Wextra -pedantic -std=c99'`

Pour activer l'alias dans le terminal courant au lieu de devoir sans arrêt le refermer/rouvrir à chaque alias ajouté ou modifié on peut utiliser la syntaxe suivante :

`source ~/.bash_aliases`

Ou de façon complètement équivalente :

`. ~/.bash_aliases`

### **1.1.3 Gérer des archives**

Créer une archive compressée en gzip d'un ensemble de fichiers :  
tar czf target.tar.gz file1 file2 file3

Extraire une archive (compressée ou non) :  
tar xf source.tar[.gz|.bz2|.xz]

Extraire une archive (compressée ou non) dans un dossier donné :  
tar xf source.tar[.gz|.bz2|.xz] -C directory

Lister le contenu d'une archive :  
tar tvf source.tar

## **Allocation Dynamique**

Utilisation de valgrind comme suit (entre crochet les options) :  
valgrind [-leak-check=full | options de valgrind] nom\_executable [options du programme]

# Chapitre 2

## Les chaînes de caractère en C

### Propriétés des chaînes de caractère en C

#### 2.1.1 Le Null-Byte

Toute chaîne se termine par un caractère nul '\0' d'ordinaire invisible car implicite. Par exemple "chien" est en réalité [c', 'h', 'i', 'e', 'n', '\0']. On appelle ce caractère null-byte en anglais.

Si ce caractère est absent, par exemple lorsqu'on construit caractère par caractère un tableau de caractères, alors les fonctions de manipulations standardes des chaînes auront des résultats imprévisibles (ces-dernières liront toute la RAM jusqu'à trouver quelque-part un null-byte ou une interdiction de lecture.)

#### 2.1.2 Mutabilité, non mutabilité de la chaîne

Il y a deux manières de déclarer une chaîne de caractère :

```
1 char s2 [] = "chiot";
2 char* s1 = "chien";
```

Dans le premier cas la chaîne est non mutable (ie. on ne peut pas modifier un ou des caractères de la chaîne mais on peut y accéder en itérant). En effet, lors de la création de la chaîne de caractère une zone mémoire est allouée (dans le stack, pas dynamiquement) avec un null-byte en fin de chaîne. La zone mémoire du premier caractère est alors renvoyé. Cependant, l'allocation ici se fait dans une zone mémoire en lecture seule. On ne peut donc pas changer la chaîne comme on veut (à moins de lui en réaffecter une nouvelle).

Dans le second cas la chaîne est mutable. Le null-byte est toujours affecté implicitement en bout de chaîne (si on vérifie la taille s2 a une taille de 6 caractères...) s2 pointe toujours vers la zone mémoire du premier caractère mais cette fois-ci la chaîne est n'est plus en lecture seule.

Dans le cas de s1 il n'y a donc aucune différence à écrire `char* s1 = "chien"` ou `const char* s1 = "chien"` !

Bien évidemment si on alloue auparavant dynamiquement la zone mémoire pour s1 et qu'on lui affecte une chaîne comme précédemment, la chaîne est alors mutable (une variable dans le tas, aka 'heap', est *toujours*<sup>?</sup> en lecture/écriture).

## Fonctions de manipulations des chaînes

### 2.2.1 Manipulations

```
#include <string.h>
char *strcat(char *dest, const char *src);
```

int strcmp(const char \*s1, const char \*s2); → à utiliser au lieu de '==' ! Avec == on compare l'égalité de référence des chaînes et non de contenu !

```
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
size_t strlen(const char *s);
char *strstr(const char *haystack, const char *needle);
```

### 2.2.2 Conversions

```
#include <stdio.h>
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

## Taille de buffers

Exemple d'utilisation de snprintf pour allouer dynamiquement des variables de tout type à la bonne taille :

```
1  char * strFormat()
2  {
3      float r1 = 3.14159f;
4      double r2 = 1.0 / 3.0;
5      const char *format = "est-ce que %g est plus grand que %lf ?";
6      char *res;
7
8      int format_len = snprintf(NULL, 0, format, r1, r2) + 1;
9      //+1 pour permettre l'écriture du null byte '\0'...
10
11     res = (char *) malloc(format_len * sizeof(char));
12     sprintf(res, format, r1, r2);
13
14     return res;
15 }
```

# Chapitre 3

## Modules

### Notions importantes pour la conception

#### 3.1.1 Déclarer vs Définir

Déclarer = "être défini ailleurs"

Définir = "créer"

On peut donc déclarer plusieurs fois une variable ou une fonction (ou toute autre chose) mais on ne peut la définir qu'une seule fois.

Une variable locale est toujours une définition. Si aucune valeur n'est renseignée alors la valeur par défaut 0 lui est donnée par le compilateur (dans le doute il vaut mieux renseigner soi-même la valeur par défaut.)

Une variable globale non initialisée peut être une déclaration ou une définition, on ne peut pas vraiment savoir quel sera le choix du compilateur à l'avance. Si on veut désambiguer cela on dispose du mot-clef 'extern'.

#### 3.1.2 .c vs .h

Le .c est écrit par le programmeur

Le .h est l'interface publique que peut lire l'utilisateur du module. Généralement cet utilisateur n'aura pas accès au .c mais au .o correspondant (illisible donc) ceci afin d'éviter que celui-ci ne modifie des fonctions et ne crée des erreurs qui n'auraient pas eu lieu d'ordinaire (l'implémentation est protégée.)

Le .h contient donc généralement des commentaires décrivant ce que font les fonctions.  
Le .c contient des commentaires expliquant l'implantation.

Théoriquement le .h devrait être écrit avant le .c

#### 3.1.3 Protéger des inclusions multiples

Directives préprocesseurs suivantes dans le .h (remarquer la convention majuscule + underscore + H :

```
1 #ifndef TOTO_H
2 #define TOTO_H
3
4 #endif
```

### 3.1.4 Comment éviter les inclusions croisées ?

Créer un nouveau .c/.h qu'on inclut dans les modules problématiques est une solution. Mais il y a peut-être mieux !

### 3.1.5 Initialisations vs Affectations

Initialiser = "mettre une valeur lors de la définition" Affecter = "remplacer la valeur courante par une autre"

## Le mot-clef static

Sur une fonction : rend la fonction locale au .c (ce qui est l'équivalent d'une méthode privée en orienté-objet).

Sur une variable globale : Une variable globale est d'ordinaire persistante et connue de tous les .c . Avec le mot-clef static elle est locale à son fichier.

Sur une variable locale : Une variable locale à une fonction est d'ordinaire non persistante. Avec le mot-clef static, cette dernière devient persistante. La valeur persistante de la variable est alors celle donnée à l'initialisation. Une variable locale static est donc toujours initialisée (sinon ça n'a pas d'utilité).

Une variable locale static est par exemple utilisée pour déclencher un événement lorsqu'on passe pour la première fois dans une fonction, exemple :

```
1 void monexemple()
2 {
3     static bool firstTime = true;
4
5     if(firstTime)
6     {
7         printf("Youhou! First time!\n");
8         firstTime = false;
9     }
10
11    blabla
12
13    ...
14
15 }
```

## Le mot-clef extern

Comme dit dans la section "Déclarer vs Définir" : une variable globale non initialisée peut être une déclaration ou une définition, on ne peut pas vraiment savoir quel sera le choix du compilateur à l'avance. Si on veut désambiguer cela on dispose du mot-clef 'extern'.

Exemple d'utilité : On veut exposer dans un .h une variable que l'utilisateur peut manipuler à sa guise mais qui est définie (ie. affectée) avec une valeur autre que 0 (cf. valeur par défaut) dans le .c (implémentation protégée).

## Chapitre 4

# Fonctions de gestions de fichiers en C

### Prog Système et bonnes pratiques pour rester zen

Toujours tester les valeurs de retours avec des assert!!!

Donc en plus des autres include utiles :

```
#include <assert.h>
```

### Les fonctions de manipulations de fichiers

#### 4.2.1 Haut-niveau (bufferisé)

Le mode bufferisé fait que les données ne sont pas immédiatement écrites dans le fichier (donc sur le disque) mais que le système les place d'abord dans une zone mémoire (un buffer).

L'intérêt de ces fonctions n'est pas explicité dans le cours (probablement une question d'efficacité, une écriture sur disque prenant du temps). Ce qu'on sait c'est qu'un arrêt brutal du programme permet, par exemple, d'éviter que des données partielles ne soient écrites (ce que l'on peut peut-être vouloir... Ou pas). En effet, celle-ci étant encore en mémoire (sauf si on a utilisé le fflush() ci-dessous) elles n'ont pas encore été écrites sur le disque et donc n'y sont pas présentes après l'arrêt.

Voici quelques fonctions de gestion de fichiers bufferisées :

```
#include <stdio.h>

— FILE *fopen(const char *pathname, const char *mode) : ouverture
— int fclose(FILE *stream) : fermeture
— size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream) : écriture en binaire
— size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream) : lecture en binaire
— int fprintf(FILE *stream, const char *format, ...) : écriture en mode texte
— fscanf(FILE *stream, const char *format, ...) : lecture en mode texte
— int fseek(FILE *stream, long offset, int whence) : se déplacer dans le fichier
— int fflush(FILE *stream) : forcer l'écriture du buffer sur disque
```

#### 4.2.2 Bas-niveau (non bufferisé)

Les fonctions de gestion de fichiers non bufferisées écrivent/lisent directement dans le fichier (pas de zone mémoire temporaire, ie. pas de buffer) :

Pour open() :

```
#include <sys/stat.h> #include <fcntl.h>
```

Pour close(), write(), read(), lseek() :

```
#include <unistd.h>
Pour lseek() :
#include <sys/types.h>
Pour dprintf() :
#include <stdio.h>

— int open(const char *path, int oflag, ...) : ouverture
— int close(int fildes) : fermeture
— ssize_t write(int fildes, const void *buf, size_t nbytes) : écriture en binaire
— ssize_t read(int fildes, void *buf, size_t nbytes) : lecture en binaire
— int dprintf(int fd, const char *format, ...) : écriture en mode texte
— off_t lseek(int fd, off_t offset, int whence) : se déplacer dans le fichier
```

**Note** : fildes = file descriptor.

## Exemple de module pour la manipulation de fichiers

```
1 void myread( int fd , void *buf , size_t nbytes )
2 {
3     int read_ret = read(fd , buf , nbytes );
4     myassert(read_ret != -1, "Erreur au niveau du myread (retour à -1)");
5 }
6
7 void mywrite( int fd , void *buf , size_t nbytes )
8 {
9     int write_ret = write(fd , buf , nbytes );
10    myassert(write_ret != -1, "Erreur au niveau du mywrite (retour à -1)");
11 }
12
13 void myclose( int fd )
14 {
15     int close_ret = close(fd );
16     myassert(close_ret != -1, "Erreur au niveau du myclose (retour à -1)");
17 }
18
19 int myopenWRO( const char* path )
20 {
21     int fd = open(path , O_CREAT | O_WRONLY, 00644);
22     myassert(fd != -1, "Erreur au niveau du myopenWRO (retour à -1)");
23     return fd ;
24 }
25
26 int myopenRDO( const char* path )
27 {
28     int fd = open(path , O_RDONLY, 00644);
29     myassert(fd != -1, "Erreur au niveau du myopenRDO (retour à -1)");
30     return fd ;
31 }
```

## **Ecriture de structures dans un fichier et lecture**

Si, dans une structure, un champ contient un pointeur (ie. une adresse vers une zone mémoire allouée dynamiquement, par exemple) alors il ne faut pas écrire directement la structure (ce qu'on peut faire si aucune champ n'est alloué dynamiquement) mais champ par champ.

Dans la zone allouée, procéder champ par champ également (et ainsi de suite si d'autres pointeurs dans la zone pointée...)

# Chapitre 5

## Abstraction pointeur

### Principe

Un pointeur occupe toujours une place 8 octets, peu importe ce vers quoi il pointe.

Partant de ce principe, le compilateur n'a pas besoin, au moment de la compilation, de savoir vers quoi les pointeurs pointent... Il sait comment traduire les demandes de mémoire (de tailles données) aux différents paramètres **dans les .c respectifs** et c'est tout ce qui lui importe.

Aussi on peut, dans le .h visible de l'utilisateur (cf. Synthèse et Analyse sur les modules), simplement déclarer la structure qu'on veut et fournir tout un tas de fonctions pour manipuler cette structure grâce à des pointeurs vers celle-ci.

Ainsi on obtient un style de programmation très proche de l'orienté objet !

### Exemple de mise en oeuvre

```
1 struct CollectionP ;
2 typedef struct CollectionP* Collection ;
3 typedef const struct CollectionP* const_Collection ;
```

**Listing 5.1** – Dans le .h visible par l'utilisateur

```
1 struct CollectionP
2 {
3     Voiture* voiTab ;
4     int voiTabSize ;
5     bool isSorted ;
6 };
```

**Listing 5.2** – Dans le .c caché de l'utilisateur

# Chapitre 6

## Tubes, Forks et Exec

### Forks/Execs

#### 6.1.1 D'abord le fork()...

Un processus est un ensemble d'instructions en train de s'exécuter consécutivement comme un chef cuisinier exécuterait sa recette :

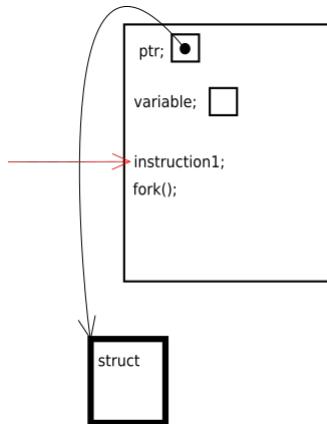


FIGURE 6.1 – Un processus

Lorsqu'on appelle un `fork()` ce processus est dupliqué en profondeur. Autrement dit, c'est comme si nous avions à présent deux chefs cuisiniers au même endroit, au même moment avec les mêmes ingrédients en même quantité (chacun ayant ses propres ingrédients).

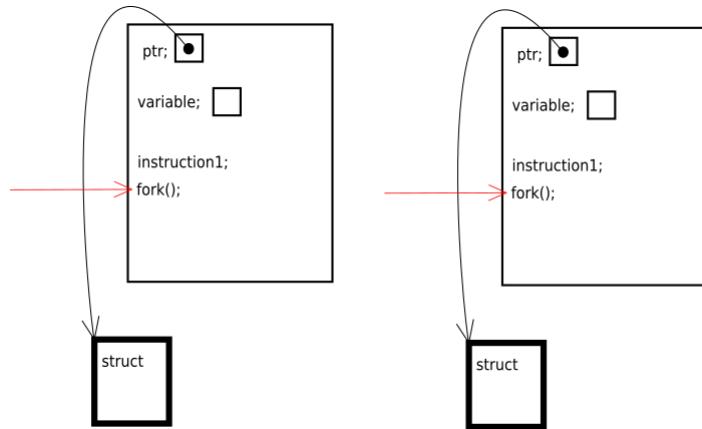
Un `fork` renvoie la valeur 0 quand il s'agit du fils. Aussi, il est aisément de faire un '`if(fork() == 0)`' et d'exécuter dans le '`if`' du code que seul le cuisiné cloné doit exécuter.

Ce code que le cuisiné cloné doit exécuter seul peut être écrit en dur dans le `if` (mais si c'est long ça peut vite devenir illisible, ou dans une fonction du même fichier .c (si ça a du sens) ou bien un exécutable quelconque venant de l'extérieur ! Ce dernier point est faisable grâce à la famille de commande `Exec`.

#### 6.1.2 Puis le Exec !

La syntaxe est généralement la suivante :

```
1 exec(chemin_vers_executable , argv);
```



**FIGURE 6.2 – Après le fork**

Il y a des nuances selon si on utilise execv, execvp, execl, execlp (cf. cours de L2).

argv est toujours construit comme suit :

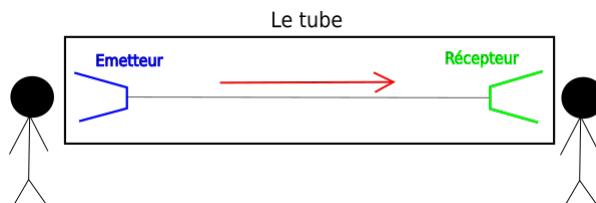
```
1 char* argv [] = {chemin_vers_executable, "arg1", "arg2", NULL};
```

On note le NULL terminal et le chemin\_vers\_executable en premier comme des conventions exigées par la famille de commandes Exec

Des conversions sont donc parfois nécessaires si on veut transmettre, par exemple, des entiers à l'exécutable appelé. Ces conversions peuvent faire appel à *sprintf(dest, format, src)* pour passer par exemple d'un int à un char[] et *long int dest = strtol(src, NULL, 10)* pour faire le chemin inverse...

## Tubes

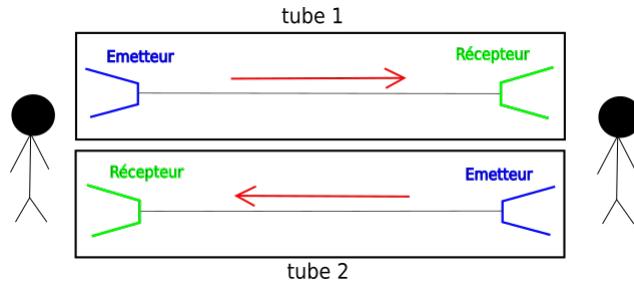
### 6.2.1 Principe



**FIGURE 6.3 – Un tube**

Un tube est semblable à un téléphone. Il permet à un processus de communiquer (ie. transmettre des informations, des données) avec un autre. Seulement, la communication peut se faire que dans un seul sens. Comme si un processus ne possédait que le micro pour parler et l'autre l'enceinte pour écouter.

Pour une communication bidirectionnelle il faudra donc un second tube comme dans ce schéma ci-dessous :



**FIGURE 6.4 – Deux tubes pour une communication bidirectionnelle**

On notera également les propriétés suivantes :

1. Si un émetteur décroche et que personne n'écoute il attend un récepteur
2. Si un récepteur décroche et que personne ne parle il attend un émetteur
3. Si un émetteur parle et que le récepteur raccroche il raccroche
4. Si un récepteur écoute que l'émetteur raccroche il raccroche

### 6.2.2 Tubes nommés

Il s'agit simplement de tubes avec des "étiquettes" les nommant. C'est-à-dire des chaînes de caractère.

Sur le système ils apparaissent comme des fichiers, ils peuvent dans une certaine mesure se manipuler comme des fichiers mais ne sont pas des fichiers (on ne peut pas écrire dedans par exemple et espérer y trouver directement du texte).

Pour les utiliser on a le jeu de commandes suivant :

1. mkfifo(tubeName, droits en octal)
2. int fdw = open(tubeName, O\_WRONLY) → on "décroche", prêt à parler
3. int fdr = open(readTube, O\_RDONLY) → on "décroche", prêt à écouter
4. On écrit dans le tube (on parle)
5. On lit dans le tube (on écoute)
6. close(fd) → on "raccroche"
7. unlink(tubeName) → on détruit le tube (le "téléphone")

### 6.2.3 Tubes anonymes

Ces "téléphones" n'ont pas d'étiquette. D'où le fait qu'ils soient anonymes. Ils n'apparaissent donc pas comme des fichiers bien qu'ils existent dans le système.

Exemple d'utilisation basique (ici le père écoute son fils mais ne lui parle pas) :

```

1 int main()
2 {
3     int fds[2];
4     int pipe_ret = pipe(fds);
5     assert(pipe_ret != -1);
6
7     if (fork() == 0)
8         fils(fds);

```

```

9
10    else
11    {
12        pere( fds );
13        wait( NULL );
14    }
15
16    return EXIT_SUCCESS;
17}
18
19 void pere( int fds [] )
20{
21    close( fds [1] );
22
23    int str_len;
24    int read_ret = read( fds [0] , &str_len , sizeof( int ) );
25    assert( read_ret != -1 );
26
27    char str[ str_len ];
28    for( int i = 0; i < str_len; i++ )
29    {
30        read_ret = read( fds [0] , &(str [i]) , sizeof( char ) );
31        assert( read_ret != -1 );
32    }
33
34    fprintf( stdout , "%s\n" , str );
35
36    close( fds [0] );
37}
38
39 void fils( int fds [] )
40{
41    close( fds [0] );
42
43    char str [] = "Bonjour!";
44    int str_len = strlen( str ) + 1; //+1 pour le '\0'
45
46    int write_ret = write( fds [1] , &str_len , sizeof( int ) );
47    assert( write_ret != -1 );
48
49    for( int i = 0; i < str_len; i++ )
50    {
51        write_ret = write( fds [1] , &(str [i]) , sizeof( char ) );
52        assert( write_ret != -1 );
53    }
54
55    close( fds [1] );
56}

```

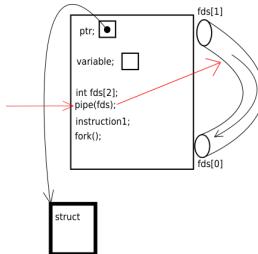
Certaines instructions (comme les close() en début de père et de fils) sont expliquées dans la sous-section suivante...

#### 6.2.4 Tubes vs Fork/Exec

Comme indiqué dans le schéma précédent il faut créer deux tubes. Un pour parler (réciproquement pour écouter côté récepteur), l'autre pour écouter (réciproquement pour parler côté récepteur).

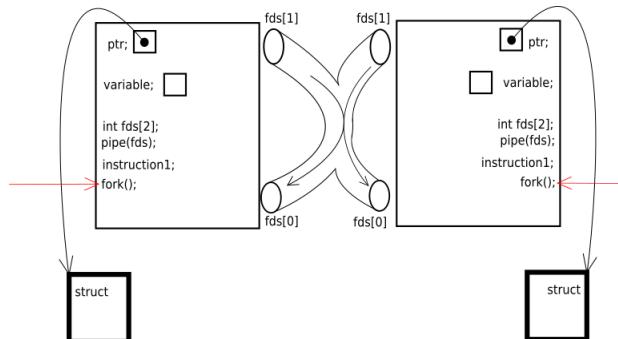
Pour les tubes nommés il n'y a pas plus de difficulté à ça (il faut simplement être vigilant à comment on décroche (en écoute vs en émission).

Pour les tubes anonymes une difficulté supplémentaire s'ajoute dans le cas d'un fork/exec... Nous avons vu dans la section précédente sur les fork/exec que tout était copié en profondeur (y compris les allocations dynamiques). Donc une variable ou zone allouée dynamiquement n'est pas la même entre le processus père/fils. Ceci est vrai sauf pour les descripteurs de fichiers (**file descriptor** en anglais, "fildes" pour Bash, "fd" ici.) On a donc le problème suivant :



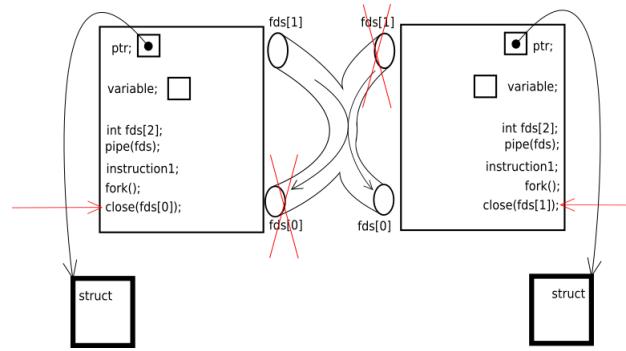
**FIGURE 6.5 – Un tube anonyme avant le fork**

Quand un tube anonyme est créé son propriété a l'émetteur (`fds[1]`) et le récepteur (`fds[0]`). Il peut donc se parler à lui-même mais ça n'a pas grand intérêt...



**FIGURE 6.6 – Un tube anonyme après le fork**

Comme les *file descriptor* sont partagés entre un processus et son clone, le processus fils dispose également de l'émetteur et du récepteur. Or on aimeraît que le père (ou le fils) ne reçoivent pas également les informations qu'il transmet ! Sinon cela pourrait créer des bugs...



**FIGURE 6.7 – Un tube anonyme après un fork puis close**

On ferme donc les émetteurs/récepteurs selon le sens de communication qu'on souhaite avoir.

Lors d'une communication bidirectionnelle avec deux tubes anonymes il faut alors faire la même chose pour les deux tubes !

# Chapitre 7

## Sémaphores

### Principe

Un sémaphore est comme un sac de jetons. Ces jetons sont pris ou donnés par des processus (cf. en utilisant des `semop()` par exemple). Le (ou les) sac(s) de jetons sont généralement initialisé (cf. `semget()` et `semctl()`) par un processus parent.

Ces jetons servent (dans notre cas) à contrôler la progression d'un processus. En forçant un processus à attendre, par exemple, qu'un autre processus remette un jeton dans le sac, par exemple, on peut protéger l'accès à des zones mémoires communes entre processus ou encore mettre en place des protocoles de communication.

Par exemple, on peut faire en sorte qu'un processus P1 ne puisse pas décrocher avant qu'un autre processus P2 n'est d'abord raccroché avec un autre processus P3 puis de nouveau décroché pour une nouvelle conversation avec P1 !

### Exemple de module pour gérer les sémaphores

```
1 struct SemaphoreP
2 {
3     int nbSem;
4     int semId;
5 };
6
7 /**
8 * Constructeurs , Destructeurs
9 */
10
11 //Un sémaphore privé est local à l'endroit où il a été défini:
12 Semaphore sema_creerPrivate(int nbSem, int droits)
13 {
14     int semid = semget(IPC_PRIVATE, nbSem, droits|IPC_CREAT|IPC_EXCL);
15     myassert(semid != -1, "Erreur : sémaphore non ou mal créée");
16
17     Semaphore res = (Semaphore) malloc(sizeof(struct SemaphoreP));
18     res->nbSem = nbSem;
19     res->semId = semid;
20     return res;
21 }
```

```

22
23 Semaphore sema_creerNouveau(const char *filename, int idProj, int droits,
24   int nbSem)
25 {
26     key_t mykey = ftok(filename, idProj);
27     myassert(mykey != -1, "Erreur : clé de sémaphore problématique");
28
29     int semid = semget(mykey, nbSem, droits | IPC_CREAT | IPC_EXCL);
30     myassert(semid != -1, "Erreur : sémaphore non ou mal créée");
31
32     Semaphore res = (Semaphore) malloc(sizeof(struct SemaphoreP));
33     res->nbSem = nbSem;
34     res->semId = semid;
35     return res;
36 }
37
38 Semaphore sema_creerExistant(const char *filename, int idProj, int nbSem)
39 {
40     // Récupérer la file :
41     key_t mykey = ftok(filename, idProj);
42     myassert(mykey != -1, "Erreur : clé de sémaphore problématique");
43
44     int semid = semget(mykey, nbSem, IPC_EXCL);
45     myassert(semid != -1, "Erreur : sémaphore non ou mal créée");
46
47     Semaphore res = (Semaphore) malloc(sizeof(struct SemaphoreP));
48     res->nbSem = nbSem;
49     res->semId = semid;
50     return res;
51 }
52
53 void sema_detruireSansSupression(Semaphore *self)
54 {
55     free(*self);
56     *self = NULL;
57 }
58
59 void sema_detruireAvecSupression(Semaphore *self)
60 {
61     int semctl_ret = semctl((*self)->semId, 0, IPC_RMID);
62     myassert(semctl_ret != -1, "Erreur : sémaphore non ou mal détruite");
63     sema_detruireSansSupression(self);
64 }
65
66
67
68 /**
69 * Initialisation , Accesseurs
70 */
71 unsigned short sema_getVal(constSemaphore self, int pos)
72 {
73     myassert((pos >= 0) && (pos < self->nbSem), "numero semaphore");
74
75     int semctl_ret = semctl(self->semId, pos, GETVAL);
76     myassert(semctl_ret != -1, "Erreur : Numéro de sémaphore invalide");

```

```

77     return semctl_ret;
78 }
79
80 void sema_setVal(Semaphore self, int pos, unsigned short val)
81 {
82     myassert((pos >= 0) && (pos < self->nbSem), "numero semaphore");
83
84     int semctl_ret = semctl(self->semId, pos, SETVAL, val);
85     myassert(semctl_ret != -1, "Erreur : Problème lors de l'affectation de
86 valeur à cette sémaphore");
87
88 void sema_getVals(constSemaphore self, int nbSem, unsigned short vals[])
89 {
90     myassert(nbSem == self->nbSem, "nombre semaphores");
91
92     for(int i = 0; i < nbSem; i++)
93         vals[i] = sema_getVal(self, i);
94 }
95
96 void sema_setVals(Semaphore self, int nbSem, const unsigned short vals[])
97 {
98     myassert(nbSem == self->nbSem, "nombre semaphores");
99
100    for(int i = 0; i < nbSem; i++)
101        sema_setVal(self, i, vals[i]);
102 }
103
104 /**
105 * Opérations basiques
106 */
107
108 void sema_prendre(Semaphore self, int pos)
109 {
110     struct sembuf semb = {pos, -1, 0};
111     int semop_ret = semop(self->semId, &semb, 1);
112     myassert(semop_ret != -1, "Impossible de prendre cette sémaphore");
113 }
114
115 void sema_vendre(Semaphore self, int pos)
116 {
117     struct sembuf semb = {pos, 1, 0};
118     int semop_ret = semop(self->semId, &semb, 1);
119     myassert(semop_ret != -1, "Impossible de vendre cette sémaphore");
120 }
121
122 void sema_attendre(Semaphore self, int pos)
123 {
124     struct sembuf semb = {pos, 0, 0};
125     int semop_ret = semop(self->semId, &semb, 1);
126     myassert(semop_ret != -1, "Impossible d'attendre cette sémaphore");
127 }
128
129
130 /**
131 * Opérations à peine moins basiques

```

```

132 /**
133 void sema_diminue(Semaphore self , int pos , int delta)
134 {
135     myassert(delta > 0, "delta doit etre > 0");
136
137     struct sembuf semb = {pos, -delta, 0};
138     int semop_ret = semop(self->semId, &semb, 1);
139     myassert(semop_ret != -1, "Erreur : Problème durant diminution de sé
140     maphore");
141 }
142
143 void sema_augmente(Semaphore self , int pos , int delta)
144 {
145     myassert(delta > 0, "delta doit etre > 0");
146
147     struct sembuf semb = {pos, delta, 0};
148     int semop_ret = semop(self->semId, &semb, 1);
149     myassert(semop_ret != -1, "Erreur : Problème durant augmentation de sé
150     maphore");
151 }
152
153 void sema_modifie(Semaphore self , int pos , int delta)
154 {
155     if(delta < 0)
156     {
157         sema_diminue(self , pos , -delta);
158     }
159     else if(delta == 0)
160     {
161         sema_attendre(self , pos);
162     }
163     else
164     {
165         sema_augmente(self , pos , delta);
166     }
167 }
```

# Chapitre 8

## Threads

### Pointeurs sur fonction

Exemple :

```
int add(int* a, float b);
```

Comment faire un pointeur sur cette fonction ?

1. int ptfn(int\* a, floatb);
2. typedef int (\*ptfn)(int\* a, float b);

On peut alors finalement utiliser le pointeur comme suit :

1. ptfn monptr;
2. monptr = add;
3. monptr(&i, 3.14);

Ceci explique pourquoi on a l'argument suivant dans certains arguments des fonctions sur les threads :

void\* (\*start\_routine) (void \*) → void\* nous indique une valeur de retour qui est un pointeur vers ce que l'on veut (et non exclusivement int comme dans l'exemple précédent), (\*start\_routine) est le pointeur sur fonction comme vu dans le 2) de l'exemple précédent, (void \*) représente les arguments qui peuvent être quelconques.

Les pointeurs sur fonction sont très utiles pour faire pour passer, par exemple, des fonctions en paramètre (exemple : fonction de tri qui se baserait sur des critères pour ordonner les éléments différents. Ces critères peuvent être représentés par des fonctions dont on passerait le pointeur dans le tri. Le tri se chargerait alors de faire respecter le critère donné !)

### Manipulation les Threads

Comme t Sous gcc, option '-pthread' obligatoire pour le compilateur **ET** l'éditeur de lien.

```
#include <pthread.h>
```

- int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine) (void \*), void \*arg) : créer et lancer un thread
- int pthread\_join(pthread\_t thread, void \*\*retval) : attendre la fin d'un thread fils
- void pthread\_exit(void \*retval) : terminer (proprement) un thread (un thread se termine également proprement lorsqu'il arrive à la fin de sa fonction support)
- pthread\_t pthread\_self(void) : renvoie l'identifiant du thread courant (anologue à getpid)

## Un exemple de module

Comme toujours en programmation système on fait des assert sur les valeurs de retour à chaque appel d'une fonction système. Ceci peut, par exemple, se traduire par un module personnel comme celui-ci :

```
1 void myPthCreate(pthread_t *thread, const pthread_attr_t *attr,
2                   void *(*start_routine) (void *), void *arg)
3 {
4     int ret = pthread_create(thread, attr, start_routine, arg);
5     assert(ret == 0);
6 }
7
8
9 void myPthJoin(pthread_t thread, void **retval)
10 {
11     int ret = pthread_join(thread, retval);
12     assert(ret == 0);
13 }
```

## Passer des arguments aux threads

**Attention :**

1. D'après les arguments de la fonction `pthread_create()` on donne toujours un pointeur vers la variable. Ce qui signifie, par exemple lors d'une itération, que si une variable est modifiée/incrémente, elle est modifiée/incrémente pour tous les threads!!! Si cette incrémentation va plus vite que l'exécution des threads, alors les threads ne recevront pas les valeurs qu'on attendait.
2. Comme il s'agit d'un `void*` pour passer un argument, le type ne peut pas être contrôlé par le compilateur dans la fonction passée en pointeur. Ce qui signifie que si les types ne correspondent pas, on aura une erreur runtime!
3. Si on veut passer plusieurs paramètres à la fonction du thread il faudra utiliser un type produit (un struc).

Une solution pour le premier point : initialiser un tableau avec une valeur pour chaque thread puis passer la valeur de ce tableau en argument du thread.

## Gérer la concurrence mémoire entre threads

Utiliser les sémaphores ! Une concurrence mémoire = une section critique.

**Attention** à minimiser le temps passé dans une section critique ! Sinon on finit par séquentialiser les threads et on ruine alors tout leur intérêt ! L'intérêt des threads est en effet de pouvoir travailler en parallèle...

## Exemple d'algorithme pour l'utilisation des threads

```
1 // Structure pour passer les paramètres aux pthread:
2 typedef struct
3 {
4     int *result;
5     int nbIterations;
6     pthread_mutex_t* mutex;
7 } ThreadData;
8
9
10 // Fonction support d'un thread:
11 void* codeThread(void* arg)
12 {
13     ThreadData *data = (ThreadData *) arg;
14     int ajout = 0;
15
16     for (int i = 0; i < data->nbIterations; i++)
17         ajout++;
18
19     pthread_mutex_unlock(data->mutex);
20     (*(data->result)) += ajout;
21     pthread_mutex_lock(data->mutex);
22     return NULL;
23 }
24
25 int main()
26 {
27     // Mutex déclaré en local:
28     pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
29
30     // Variable partagée:
31     int result = 0;
32
33     // On stocke tous les identifiants des threads:
34     pthread_t tabId[NB_THREADS];
35     // Chaque thread a sa propre structure:
36     ThreadData datas[NB_THREADS];
37
38     // Pré-initialisation des données:
39     for (int i = 0; i < NB_THREADS; i++)
40     {
41         datas[i].result = &result;
42         datas[i].nbIterations = NB_ITERATIONS;
43         datas[i].mutex = &mutex;
44     }
45
46     // Lancement des threads:
47     for (int i = 0; i < NB_THREADS; i++)
48     {
49         // On passe un pointeur sur une struct différente chaque fois:
50         int ret = pthread_create(&(tabId[i]), NULL, codeThread, &(datas[i]));
51     }
52 }
```

```
53
54 // Attente de la fin des threads:
55 for (int i = 0; i < NB_THREADS; i++)
56 {
57     int ret = pthread_join(tabId[i], NULL);
58     assert(ret == 0);
59 }
60
61 // On détruit le mutex:
62 pthread_mutex_destroy(&mutex);
63
64 // Affichage résultat:
65 printf("Fin thread principal : résultat\n");
66 printf("Attendu : %d\n", NB_THREADS * NB_ITERATIONS);
67 printf("Obtenu : %d\n", result);
68
69 return EXIT_SUCCESS;
70 }
```

Deuxième partie

Analyse

## Chapitre 9

# Rappels Bash et Allocation Dynamique

### Résumé et cadre de travail

```
(base) florian@CRegJr:~$ alias  
alias mygcc='gcc -Wall -Wextra -Werror -pedantic -std=c99 -g -o'
```

Listing 9.1 – Définition de l'alias pour simplifier la syntaxe de compilation

### Bash

#### Point de cours

Vous devez maîtriser :

- Les commandes de bases suivantes : *cd*, *ls*, *cp*, *mv*, *ln*, *mkdir*, *rm*, *rmdir*, *cat*, *more/less* et beaucoup d'autres
- Les chemins absolus et relatifs
- La différence entre code source et code exécutable
- Les droits des fichiers et répertoires (avec *chmod*)
- Les redirections de base : *>*, *>>*, *<*, *2>*, *2>>*, *|*

Vous devez savoir :

- lister les processus : *ps*, *top*, *jobs*
- envoyer un signal à un processus : *kill*
- lancer les processus en arrière et avant-plan : *&*, *ctrl-z*, *bg*, *fg*

## Exercice 3 : Sempiternel Hello World

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     printf("Hello World!\n");
7     return EXIT_SUCCESS;
8 }
```

Listing 9.2 – Mon exemplaire de "Hello World"

```
1 /*
2 * Points importants
3 * - on ne met que les include strictement nécessaires
4 * - si on n'est pas capable d'expliquer la présence d'un include,
5 *   c'est qu'il est inutile et on l'enlève
6 * - on utilise les constantes nommées lorsqu'elles existent car cela
7 *   apporte du sens et facilite la lecture du programme. Donc un
8 *   "return 0" en fin de main est à proscrire
9 * - l'indentation est obligatoire et doit être la même partout
10 * - lors d'une épreuve, un code mal indenté est fortement pénalisé
11 */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15
16 int main()
17 {
18     printf("Hello World!\n");
19
20     return EXIT_SUCCESS;
21 }
```

Listing 9.3 – L'exemplaire de l'enseignant

## Exercice 4 : Archive, Compression, Somme de contrôle

### Point de cours

Voici quelques rappels histoire de fixer le vocabulaire.

Une *archive* permet de regrouper toute une arborescence dans un seul fichier, généralement pour faciliter sa manipulation (déplacement, transmission, sauvegarde, ...).

Une archive peut être compressée. C'est généralement le cas mais pas une obligation. De fait les archiveurs proposent <sup>a</sup> systématiquement une option de compression.

Il s'agit juste d'insister sur le fait qu'archivage et compression sont deux processus complètement séparés.

Enfin, lorsqu'on transmet une archive, il faut s'assurer qu'elle n'a pas été corrompue (intentionnellement ou non) lors du transfert. Il est donc possible de l'accompagner d'une somme de contrôle.

Commande *tar*, voici une liste d'options :

- c créer une archive
- x extraire une archive
- t lister le contenu d'une archive
- f nom du fichier archive à créer ou à lire
- z manipulation d'une archive compressée avec *gzip*
- j manipulation d'une archive compressée avec *bzip2*

Les options *c/x/t* sont exclusives, ainsi que les options *z/j*.

La commande *md5sum* sans option permet de calculer une somme de contrôle, et avec l'option *-c* de vérifier une somme de contrôle.

<sup>a</sup>. et parfois imposent, ce qui est discutable

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1$ cp -R ./  
code_fourni/EXO_04/PROJET_TEST/ projet_test  
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1$ ls  
code_fourni      CR      projet_test  test.tar  
code_fourni.tgz  ex3.c   test          tp_01.pdf
```

**Listing 9.4** – Copie d'un dossier dans un autre avec l'option *-R* de la commande *cp*

```

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/vide$ tar xvf projet_test
    .tar
projet_test/
projet_test/SOURCE/
projet_test/SOURCE/gloplop.c
projet_test/SOURCE/pasglop.c
projet_test/SOURCE/toto.c
projet_test/SOURCE/toto.h
projet_test/rapport.pdf
projet_test/RESSOURCES/
projet_test/RESSOURCES/data

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/vide$ ls
projet_test  projet_test.tar

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/vide$ rm -R projet_test

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/vide$ ls -lh
total 160K
-rw-rw-r-- 1 crex crex 160K sept. 20 19:27 projet_test.tar

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/vide$ bzip2 -9
    projet_test.tar

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/vide$ ls -lh
total 100K
-rw-rw-r-- 1 crex crex 98K sept. 20 19:27 projet_test.tar.bz2

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/vide$ tar xjvf
    projet_test.tar.bz2
projet_test/
projet_test/SOURCE/
projet_test/SOURCE/gloplop.c
projet_test/SOURCE/pasglop.c
projet_test/SOURCE/toto.c
projet_test/SOURCE/toto.h
projet_test/rapport.pdf
projet_test/RESSOURCES/
projet_test/RESSOURCES/data

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/vide$ ls
projet_test  projet_test.tar.bz2

```

**Listing 9.5** – Illustration de l'utilité et de comment manipuler la compression/décompression sur les archives

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/projet_test/
      SOURCE$ ls
glopglob.c  pasglob.c  toto.c  toto.h

(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/projet_test/
      SOURCE$ md5sum *.c > verif

(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/projet_test/
      SOURCE$ ls
glopglob.c  pasglob.c  toto.c  toto.h  verif

(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/projet_test/
      SOURCE$ cat verif
15de91a075434f61795d3a3f5c1c8cce  globglob.c
63b3e2d0b4f9ff38714e524a0405629e  pasglob.c
7d7ba68bf9d9845327aadca9558d717  toto.c

(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/projet_test/
      SOURCE$ md5sum -c verif
globglob.c: OK
pasglob.c: OK
toto.c: OK

(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/projet_test/
      SOURCE$ nano globglob.c

(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex4/projet_test/
      SOURCE$ md5sum -c verif
globglob.c: FAILED
pasglob.c: OK
toto.c: OK
md5sum: WARNING: 1 computed checksum did NOT match
```

**Listing 9.6** – Illustration de l’usage et de l’utilité de la commande md5sum pour créer et vérifier des sommes de contrôle

## Exercice 5 : Redirections

### 9.5.1 Un programme pour manipuler des flux d'entrées et de sorties

**Question :**

Faites un programme qui affiche 3 messages différents :

- un avec printf,
- un avec fprintf sur la sortie standard,
- un avec fprintf sur la sortie erreur.

Puis il lit un entier sur l'entrée standard, entier qui est aussitôt affiché sur la sortie standard.  
Chaque affichage est suivi d'un passage à la ligne.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     printf("Hey Jude! Don't make it bad!\n");
7     fprintf(stdout, "Hey Jude! Don't make it bad!\n");
8     fprintf(stderr, "Hey Jude! Don't make it bad!\n");
9
10    char user_input[100];
11    fscanf(stdin, "%s", user_input);
12    fprintf(stdout, "%s\n", user_input);
13
14    return EXIT_SUCCESS;
15 }
```

**Listing 9.7** – Code source de ex5.c

### 9.5.2 Manipulation sur les opérateurs de redirections > et |

**Question :** Lancez le programme avec la redirection suivante en essayant de deviner le résultat : > sur un fichier

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ ./ex5 > test.txt
Hey Jude! Don't make it bad!
Hi!

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat test.txt
Hey Jude! Don't make it bad!
Hey Jude! Don't make it bad!
Hi!
```

**Réponse :** On constate que le fprintf(stderr,...) a été affiché dans la console mais que le fprintf(stdout,...) a lui été redirigé dans le fichier test.txt.

On voit également que le premier "Hi!" correspondant à l'input que j'ai donné pour le fscanf(stdin,...) a été redirigé également dans cat test.txt (le premier "Hi!" correspondant simplement à ce que j'ai entré dans la console).

**Question :** Lancez le programme avec la redirection suivante en essayant de deviner le résultat : > sur /dev/null

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ ./ex5 > /dev/null
Hey Jude! Don't make it bad!
lo

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat /dev/null
```

**Réponse :** Tous les stdout ont été redirigé vers /dev/null qui les a effacé. /dev/null a donc permis d'ignorer le flux sortant stdout. En revanche le flux stderr a été affiché dans la console.

**Question :** Lancez le programme avec la redirection suivante en essayant de deviner le résultat : > sur la commande less ( ??? pourquoi cela n'a-t-il aucun sens ?)

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ ./ex5 > less
Hey Jude! Don't make it bad!
Hi!

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat less
Hey Jude! Don't make it bad!
Hey Jude! Don't make it bad!
Hi!

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ ls
ex5  ex5.c  less  test.txt
```

**Réponse :** L'intention première était donc de rediriger la sortie stdout (sortie 1) sur la commande less. Cependant l'opérateur > sert à rediriger vers des fichiers. Ne trouvant pas de fichier nommé "less" dans mon répertoire, le système en a créé un et y a redirigé le flux...

**Question :** Lancez le programme avec la redirection suivante en essayant de deviner le résultat : 2> sur un fichier

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ ./ex5 2> test.txt
Hey Jude! Don't make it bad!
Hey Jude! Don't make it bad!
pol
pol

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat test.txt
Hey Jude! Don't make it bad!
```

**Réponse :** Cette fois-ci les stdout sont affichés dans la console (on omet le premier "pol" qui correspond à mon input) et les stderr ont été redirigés dans test.txt. On note également que > a écrasé ce qui y était auparavant écrit. Pour un ajout en fin de fichier on aurait dû utiliser l'opérateur » ...

**Question :** Lancez le programme avec la redirection suivante en essayant de deviner le résultat : 2> sur /dev/null

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ ./ex5 2> /dev/null
Hey Jude! Don't make it bad!
Hey Jude! Don't make it bad!
hi
hi

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat /dev/null
```

**Réponse :** Cette commande semble confirmer notre hypothèse précédente que /dev/null est un fichier spécial servant à ignorer des flux de sorties (stdout ou stderr).

**Question :** Lancez le programme avec la redirection suivante en essayant de deviner le résultat : > et 2> sur deux fichiers distincts

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ ./ex5 > stdout.txt 2>
stderr.txt
Hi!

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ ls
ex5  ex5.c  less  stderr.txt  stdout.txt  test.txt

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat std
stderr.txt  stdout.txt

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat stdout.txt
Hey Jude! Don't make it bad!
Hey Jude! Don't make it bad!
Hi!

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat stderr.txt
Hey Jude! Don't make it bad!
```

**Réponse :** La commande a redirigé tous les stdout vers le fichier "stdout.txt" puis elle a redirigé les stderr vers le fichiers stderr.txt. L'interprétation de la commande s'est donc faite de gauche à droite et si nous avions écrit "../ex5 2 > stderr.txt > stdout.txt" elle aurait alors d'abord redirigé les stderr vers stderr.txt puis les stdout vers stdout.txt ce qui aurait produit le même résultat.

**Question :** Lancez le programme avec la redirection suivante en essayant de deviner le résultat : | sur un fichier ( ??? pourquoi cela n'a-t-il aucun sens ?)

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ ./ex5 | test.txt
Hey Jude! Don't make it bad!
test.txt: command not found
Hi!
```

**Réponse :** L'opérateur / correspond aussi à une redirection (comme >) mais à une redirection vers une commande ! Bash cherche donc une commande qui s'appellerait test.txt (en utilisant la variable d'environnement \$PATH et dans le dossier courant...) mais ici il ne la trouve pas. Si une commande test.txt le flux aurait par contre été redirigé vers cette commande qui s'en serait servi ou pas (selon son code source...)

**Question :** Lancez le programme avec la redirection suivante en essayant de deviner le résultat :  
| sur la commande less

**Réponse :** Dans ce dernier cas, la commande less est lancée ce qui nous amène à un écran noir. En effet, less attend l'input pour fscanf. Une fois l'input reçu, less traite tous les flux stdout en les présentant sous forme de pages comme elle sait le faire. Quand on quitte la commande less on découvre alors que les stderr ont, eux, été affichés dans la console d'origine

### 9.5.3 Quelques exercices d'anticipation sur les résultats de redirections

**Question :** Pourquoi la commande suivante n'a que peu d'intérêt :  
\$ ls > result | less

**Réponse :** On crée un fichier "result" (cf. on utilise >) dans lequel on écrit le flux stdout de la commande "ls". Ce qui était dans stdout n'y est plus car on l'a utilisé dans le fichier result. Pourtant on essaye ensuite de brancher un stdout sur le stdin de la commande less. result est un fichier et n'a pas de stdot. Le stdot de ls a été redirigé. less n'a pas de stdot en entrée et n'a donc rien à afficher.

**Question :** De même pour la commande :  
\$ ls | less > result

**Réponse :** Le stdot de ls est passé en stdin de less (donc en argument), cependant l'affichage de less (qui est son stdot) est redirigé vers le fichier result. De nouveau on n'a donc aucun affichage (normalement les stdot sont redirigés vers les tty, ie. nos terminaux). Il faut ouvrir le fichier result pour lire le résultat de ls qui aurait aussi bien pu être affiché dans le terminal ou se passer de less. Pas très utile...

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ rm result
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ ls | less > result
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat result
erreurs
ex5
ex5.c
less
result
stderr.txt
stdout.txt
test.txt
```

Listing 9.8 – Test de la commande ls | less > result

**Question :** Pourquoi la commande suivante est beaucoup plus intéressante :  
\$ ls 2> erreurs | less

**Réponse :** Ici on fait deux choses... Premièrement on redirige toutes les erreurs (donc le flux stderr) de ls vers un fichier appelé "erreurs". Puis on redirige le flux stdot de ls vers less. On a donc simultanément le résultat de ls paginé par less et stocké les erreurs de ls.

**Question :** Quelles sont les différences entre les deux commandes suivantes ?  
\$ ls > result 2> erreurs  
\$ ls 2> erreurs > result

**Réponse :** Aucune

*Manipulation personnelle n°1 :*

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat test.txt  
  
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat stderr.txt  
Hey Jude! Don't make it bad!  
  
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ stderr.txt > test.txt  
stderr.txt: command not found  
  
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ ls > stderr.txt > test.txt  
  
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex5$ cat test.txt  
erreurs  
ex5  
ex5.c  
less  
result  
stderr.txt  
stdout.txt  
test.txt
```

**Réponse :** Le `stdout` de `ls` est redirigé vers `stderr.txt`. `stderr.txt` devient le nouveau `stdout`<sup>1</sup> et prend donc la valeur du `stdout` de `ls`. Ce `stdout` est alors finalement redirigé vers `test.txt`. Ce-dernier n'est pas lui-même redirigé donc la valeur finit par être copiée dans `test.txt`. Comme précédemment, `stdout` étant redirigé, la destination précédente du `stdout` (à l'origine le `tty`, le terminal...) est vidée.

#### 9.5.4 Rediriger une sortie erreur dans un tube ?

**Question :** Comment fait-on pour rediriger la sortie erreur dans un tube (i.e. avec un `|`) ?

**Réponse :** Réponse proposée : Pas de réponse

---

1. Pipe is used to pass `stdout` to another program or utility `stdin`. Redirect is used to pass `stdout/stderr` to either a file or stream

## Exercice 6 : Variables Shell et Programmes C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* Écrivez un programme, sans regarder la solution fournie, qui affiche :
5 - le nombre et la liste des arguments passés en ligne de commande
6 - le contenu du tableau env
7 Écrivez un code le plus lisible possible.
8 Comparez votre code avec les trois versions fournies. */
9
10 int main(int argc, char *argv[], char *env[])
11 {
12     fprintf(stdout, "There are %i arguments\n\n", argv);
13
14
15     fprintf(stdout, "The arguments are:\n");
16     for(int i = 0; i < argc; i++)
17         fprintf(stdout, "%i : %s\n", i, argv[i]);
18
19
20     fprintf(stdout, "\n\n The env content is: \n");
21
22     int j = 0;
23     while(env[j] != NULL)
24     {
25         fprintf(stdout, "%i : %s\n", j, env[j]);
26         j++;
27     }
28
29     return EXIT_SUCCESS;
30 }
```

Listing 9.9 – Mon code source du programme répondant à la consigne en commentaire (cf. texte en rouge)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main( int argc , char * argv[] , char * env[] )
5 {
6     int i;
7
8     printf("Il y a %d paramètre(s)\n" , argc -1);
9     for (int i = 1; i < argc; i++)
10         printf(" - %s\n" , argv[ i ] );
11     printf("\n");
12
13     printf("Voici les variables d'environnement :\n");
14     i = 0;
15     while (env[ i ] != NULL)
16     {
17         printf(" - %s\n" , env[ i ] );
18         i++;
19     }
20
21     return EXIT_SUCCESS;
22 }
```

**Listing 9.10** – Code source de l'enseignant

## Exercice 7 : Variables Shell et Programmes C (suite)

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main( int argc , char *argv [] )
5 {
6     if( argc != 2 )
7     {
8         fprintf(stderr , "One argument (only) needed!\n");
9         return EXIT_FAILURE;
10    }
11
12    else
13    {
14        char *env_var = getenv(argv[1]);
15
16        if( env_var == NULL )
17        {
18            printf("Environment variable doesn't exist!\n");
19            return EXIT_SUCCESS;
20        }
21
22        else
23        {
24            printf("%s\n", env_var);
25            return EXIT_SUCCESS;
26        }
27    }
28 }
```

**Listing 9.11** – Un programme pour tester l'existence de variables d'environnement

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ ./ex7
One argument (only) needed!

(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ ./ex7 TOTO
Environment variable doesn't exist!

(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ ./ex7 USER
florian
```

**Listing 9.12** – Résultat du programme sur quelques exemples...

**Question :** Pourquoi le programme dit-il que la variable FOO n'existe pas ?

**Réponse :** En utilisant le programme réalisé lors de l'exercice 6 on se rend compte que la variable FOO. Pourtant selon l'environnement elle existe :

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ echo $FOO
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ FOO=13
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ echo $FOO
13
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ ./ex7 FOO
Environment variable doesn't exist!
```

**Listing 9.13** – Résultat du programme sur une variable existante de Bash et modifiée dans le terminal

**Réponse :** Il est donc possible que certaines variables d'environnement ne sont pas prises en compte par `getenv()`. Pour qu'elle les prenne en compte on peut utiliser :

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ export FOO  
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ ./ex7 FOO  
13
```

**Réponse :** Cette fois-ci la variable a donc été visible pour le programme. En effet, `export` est une fonction builtin du Bash. Comme pour toutes les fonctions builtin on ne les consulte pas avec la commande "man" mais avec l'option `-help` :

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ export --  
      help  
export: export [-fn] [name[=value] ...] or export -p  
      Set export attribute for shell variables.  
  
Marks each NAME for automatic export to the environment of subsequently  
executed commands. If VALUE is supplied, assign VALUE before exporting  
  
Options:  
-f          refer to shell functions  
-n          remove the export property from each NAME  
-p          display a list of all exported variables and functions  
  
An argument of '--' disables further option processing.  
  
Exit Status:  
Returns success unless an invalid option is given or NAME is invalid.
```

**Réponse :** On constate alors que chaque variable d'environnement a un attribut 'export' qui peut être activé par la commande `export`.

**Question :** Créez une variable BAR initialisée à 15 pour que votre programme la détecte.

Lancez une autre console, et exécutez votre programme toujours sur la variable BAR. Pourquoi n'est-elle plus visible ?

Que faudrait-il faire pour que cette variable soit définie automatiquement dans toutes les consoles ?

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ ./ex7 BAR  
Environment variable doesn't exist!  
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ BAR=15  
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ export BAR
```

Dans un autre terminal on a alors :

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex7$ ./ex7 BAR  
Environment variable doesn't exist!
```

**Réponse :** La variable n'est plus visible car elle est définie localement au terminal en cours. Pour que cette variable soit définie automatiquement dans toutes les consoles il faut écrire l'export dans le fichier `.bashrc` qui est dans le `$HOME`... On peut de même y définir `mygcc` comme défini dans l'énoncé (même si sur la plupart des distributions linux ils recommandent de le faire dans le dossier `.bash_aliases` toujours dans le `$HOME`).

## Exercice 8 : Relais

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
{
5     int copie = fgetc(stdin);
6     while(copie != EOF)
7     {
8         fputc(copie, stdout);
9         copie = fgetc(stdin);
10    }
11
12
13    return EXIT_SUCCESS;
14 }
```

**Listing 9.14** – Programme copiant des caractères un par un

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex8$ nano src
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex8$ ./ex8 < src
> target
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex8$ ls
ex8  ex8.c  src  target
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex8$ cat target
lorem ipsum lorem ipsum lorem ips... Hum?
```

**Listing 9.15** – Illustration du fonctionnement du programme dans Bash

## Exercice 9 : B.a.-ba

```
1 /* Écrivez une fonction qui prend deux réels en paramètres et retourne
2 la somme des dits paramètres. Testez cette fonction, dans le main, en
3 affichant la somme de 1.12032002 et 2.02127003.*/
4
5 #include <stdlib.h>
6 #include <stdio.h>
7
8 int main( int argv , char *argc [] )
9 {
10     if( argv != 3 )
11     {
12         printf("Wrong number of arguments (2 required)\n");
13         return EXIT_SUCCESS;
14     }
15
16     else
17     {
18         float v1 = atof( argc[1] );
19         float v2 = atof( argc[2] );
20         float sum = v1 + v2;
21         printf("The sum is: %f\n" , sum );
22         return EXIT_SUCCESS;
23     }
24 }
```

**Listing 9.16** – Programme permettant de faire la somme de deux flottants dans Bash

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex9$ ./ex9
1.12032002 2.02127003
The sum is: 3.141590
```

**Listing 9.17** – Résultat du programme permettant de faire la somme de deux flottants dans Bash

## Exercice 10 : Utilisation des Antiquotes

**Question :** Essayez les deux commandes suivantes en essayant de deviner ce qu'elles font (attention le mot tty est entouré par des anti-quotes).

```
$ echo "Bonjour"  
$ echo "Bonjour" > 'tty'
```

Expliquez.

De même essayez les commandes :

```
$ date  
$ echo 'date'  
$ echo 'date'
```

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1$ date  
lun. 21 sept. 2020 11:06:21 CEST  
  
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1$ echo 'date'  
lun. 21 sept. 2020 11:06:27 CEST  
  
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1$ echo 'date'  
date
```

**Question :** Pour différenciez l'utilisation des guillemets et des quotes, essayez les commandes suivantes :

```
$ echo 'Bonjour $toto'  
$ echo "Bonjour $toto"  
$ echo 'Bonjour $USER'  
$ echo "Bonjour $USER"
```

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1$ echo 'Bonjour $toto'  
Bonjour $toto  
  
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1$ echo "Bonjour $toto"  
Bonjour  
  
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1$ echo 'Bonjour $USER'  
Bonjour $USER  
  
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1$ echo "Bonjour $USER"  
Bonjour crex
```

**Question :** Pourquoi la commande suivante n'a (vraisemblablement) pas de sens (attention ce sont encore des anti-quotes) ?

```
$ echo 'Bonjour $USER'
```

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1$ echo 'Bonjour $USER'  
Bonjour: command not found
```

**Réponse :** Les antiquotes servent à substituer des commandes à leurs résultats (comme `$( )`). Elles sont donc exécutées en premier lorsqu'une ligne de commande(s) Bash est parsée. Ici la commande Bonjour n'existe pas et ne peut donc pas l'appliquer à la variable d'environnement `$USER`...

**Question :** Alors que les commandes suivantes fonctionnent (ligne 1 : quotes ; ligne 2 : anti-quotes) ?

```
$ alias Bonjour='echo Bonjour'  
$ echo 'Bonjour $USER'
```

**Réponse :** On crée un alias Bonjour qui devient la commande "echo Bonjour". Son résultat est donc substituable par les antiquotes...

## Exercice 11 : Pointeurs et Allocation Dynamique

### 9.11.1 Pointeurs sans allocation

**Question :** Écrire un programme (tout le code est dans le main) qui :

- déclare une variable i entière
- l'initialise à 99
- l'affiche
- lui ajoute 2
- l'affiche

Continuez le programme qui :

- déclare une variable pi qui est un pointeur sur un entier
- l'initialise pour qu'elle pointe sur la zone mémoire de la variable i
- affiche la valeur du pointeur et de la zone pointée
- ajoute 2 à la zone pointée
- affiche la valeur du pointeur et de la zone pointée
- affiche le contenu de la variable i

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     //Première consigne...
7     int i = 99;
8     printf("%i\n", i);
9     i += 2;
10    printf("%i\n", i);
11
12    //On continue...
13    int *pi;
14    pi = &i;
15    printf("%p : %i\n", (void *) pi, *pi);
16    *pi += 2;
17    printf("%p : %i\n", (void *) pi, *pi);
18    printf("Au final i vaut: %i", i);
19
20    return EXIT_SUCCESS;
21 }
```

### 9.11.2 Oubli d'allocation

**Question :** On demande dans cet exercice un code incorrect avec un warning à la compilation qui le signale.

Dans la fonction main déclarez un pointeur sur un entier sans l'initialiser. Affichez la valeur du pointeur, puis la valeur de la zone pointée. Faites un autre programme avec exactement le même code mais dans une autre fonction autre que le main.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     int *p;
7     printf("%p : %i\n", (void *) p, *p);
8     return EXIT_SUCCESS;
9 }
```

**Listing 9.18** – Version où le pointeur 'p' est dans la fonction main

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 void toto()
5 {
6     int *p;
7     printf("%p : %i\n", (void *) p, *p);
8 }
9
10 int main()
11 {
12     toto();
13     return EXIT_SUCCESS;
14 }
```

**Listing 9.19** – Version où le pointeur 'p' est en-dehors de la fonction main

**Question :** Lancez les deux programmes et comparez.

```

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex11$ ./ex11_2a
Segmentation fault (core dumped)

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex11$ ./ex11_2b
0x5565541d4060 : -98693133
```

**Réponse :** On constate donc que dans la première version on a une erreur de segmentation (cf. Le signal SIGSEGV ci-dessous) tandis que dans la seconde version le pointeur semble avoir bien été alloué et semble avoir trouvé une valeur "poubelle" (valeur laissée par un programme précédent).

**Question :** Relancez les deux programme avec valgrind. Si votre exécutable s'appelle a.out, lancez le ainsi :

```
$ valgrind ./a.out
```

```

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex11$ valgrind ./ex11_2a
==97737== Command: ./ex11_2a
==97737==
==97737== Use of uninitialised value of size 8
==97737==    at 0x109159: main (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/TP
==97737==      /TP1/Ex11/ex11_2a)
==97737==
==97737== Invalid read of size 4
==97737==    at 0x109159: main (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/TP
==97737==      /TP1/Ex11/ex11_2a)
==97737== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==97737==
==97737==
==97737== Process terminating with default action of signal 11 (SIGSEGV)
==97737== Access not within mapped region at address 0x0
==97737==    at 0x109159: main (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/TP
==97737==      /TP1/Ex11/ex11_2a)
==97737== If you believe this happened as a result of a stack
==97737== overflow in your program's main thread (unlikely but
==97737== possible), you can try to increase the size of the
==97737== main thread stack using the --main-stacksize= flag.
==97737== The main thread stack size used in this run was 8388608.
==97737==
==97737== HEAP SUMMARY:
==97737==     in use at exit: 0 bytes in 0 blocks
==97737==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==97737==
==97737== All heap blocks were freed -- no leaks are possible
==97737==
==97737== Use --track-origins=yes to see where uninitialised values come
==97737== from
==97737== For lists of detected and suppressed errors, rerun with: -s
==97737== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)

```

**Listing 9.20** – Résultat de Valgrind sur la première version

```

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex11$ valgrind ./ex11_2b
==144056== Command: ./ex11_2b
==144056==
==144056== Use of uninitialised value of size 8
==144056==    at 0x109159: toto (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==    by 0x109187: main (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==
==144056== Conditional jump or move depends on uninitialised value(s)
==144056==    at 0x48E7C02: __vfprintf_internal (vfprintf-internal.c:1687)
==144056==    by 0x48D1EBE: printf (printf.c:33)
==144056==    by 0x109172: toto (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==    by 0x109187: main (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==
==144056== Use of uninitialised value of size 8
==144056==    at 0x48CB7BA: _itoa_word (_itoa.c:180)
==144056==    by 0x48E76F4: __vfprintf_internal (vfprintf-internal.c:1687)
==144056==    by 0x48D1EBE: printf (printf.c:33)
==144056==    by 0x109172: toto (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==    by 0x109187: main (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==
==144056== Conditional jump or move depends on uninitialised value(s)
==144056==    at 0x48CB7CC: _itoa_word (_itoa.c:180)
==144056==    by 0x48E76F4: __vfprintf_internal (vfprintf-internal.c:1687)
==144056==    by 0x48D1EBE: printf (printf.c:33)
==144056==    by 0x109172: toto (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==    by 0x109187: main (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==
==144056== Conditional jump or move depends on uninitialised value(s)
==144056==    at 0x48E83A8: __vfprintf_internal (vfprintf-internal.c:1687)
==144056==    by 0x48D1EBE: printf (printf.c:33)
==144056==    by 0x109172: toto (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==    by 0x109187: main (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==
==144056== Conditional jump or move depends on uninitialised value(s)
==144056==    at 0x48E786E: __vfprintf_internal (vfprintf-internal.c:1687)
==144056==    by 0x48D1EBE: printf (printf.c:33)
==144056==    by 0x109172: toto (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==    by 0x109187: main (in /home/crex/MEGA/Bureau/Etudes/UE3_PAC/
    TP/TP1/Ex11/ex11_2b)
==144056==
0x109060 : -98693133
==144056==
==144056== HEAP SUMMARY:
==144056==     in use at exit: 0 bytes in 0 blocks

```

```

==144056==    total heap usage: 1 allocs , 1 frees , 1,024 bytes allocated
==144056==
==144056== All heap blocks were freed — no leaks are possible
==144056==
==144056== Use --track-origins=yes to see where uninitialized values come
from
==144056== For lists of detected and suppressed errors, rerun with: -s
==144056== ERROR SUMMARY: 16 errors from 6 contexts (suppressed: 0 from 0)

```

**Listing 9.21** – Résultat de Valgrind sur la seconde version

**Réponse :** On constate dans la première version que notre pointeur non initialisé pointe vers `0x0`, soit `NULL`. En revanche, dans la seconde version, ce même pointeur non initialisé a reçu la valeur `0x109060` soit 1 octet après l'adresse de la fonction `toto()` `0x109159`.

### 9.11.3 Non-sens

**Question :** Pourquoi ce code n'a pas de sens (et d'ailleurs ne compile pas) ?

```

1 int i = 5;
2 printf("l'adresse de l'adresse de i vaut : %p\n", (void *) &(&i));

```

**Réponse :** *i* est une variable entière qui a une adresse physique et un contenu qui vaut 5. Quand on écrit `& i` on essaye d'accéder à l'adresse de *i* (ce qui est possible). Quand on fait `& (&i)` on essaye d'accéder à l'adresse physique de la variable *i*. Ce qui n'a pas de sens car il n'y a pas d'adresse à des adresses physiques, les adresses physiques sont inscrites dans le hardware de la mémoire...

### 9.11.4 Allocation de structures

**Question :** Définissez une structure qui contient :

- un float
- un tableau static de float de taille 10
- un entier qui est la taille du tableau ci-dessous
- un tableau dynamique de float

Écrivez une fonction qui initialise une telle structure. La fonction prend trois paramètres :

- la structure à initialiser
- un float
- un entier qui est la taille du second tableau

Initialisez tous les flottants à votre convenance.

Écrivez une fonction qui crée un tableau de structures. La taille du tableau est passée en paramètre (et c'est le seul paramètre) et la fonction renvoie le tableau créé. Les cases du tableau doivent être initialisées avec la fonction précédente (les paramètres sont laissés à votre choix).

Écrivez une fonction qui libère les ressources mémoire d'un tableau créé par la fonction précédente. La fonction prend comme seuls paramètres le tableau à libérer et sa taille.

Écrivez un main qui crée un tableau de 15 cases et le libère.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define TAILLE1 10
5 #define STRUCT_ARR_SIZE 15
6
7 struct Exo
8 {
9     float f;
10    float tab1[TAILLE1];
11    int taille2;
12    float *tab2;
13 };
14
15 // fonction init
16 void init_struct(struct Exo *s, float x, int n)
17 {
18     s->f = x;
19
20     for (int i = 0; i < TAILLE1; i++)
21         s->tab1[i] = (float) i;
22
23     s->taille2 = n;
24     s->tab2 = (float *) malloc(n * sizeof(float));
25 }
26
27 // fonction creer
28 struct Exo* create_struct_arr(int n)
29 {
30     struct Exo *res = (struct Exo *) malloc(n * sizeof(struct Exo));
31
32     for (int i = 0; i < n; i++)
33         init_struct(&(res[i]), 3.14, 3);
34
35     return res;
36 }
37
38 // fonction libere
39 void free_struct_arr(struct Exo *arr, int arr_size)
40 {
41     for (int i = 0; i < arr_size; i++)
42         free(arr[i].tab2);
43
44     free(arr);
45 }
46
47
48 int main()
49 {
50     // appeler creer et libere
51     struct Exo *s = create_struct_arr(STRUCT_ARR_SIZE);
52     free_struct_arr(s, STRUCT_ARR_SIZE);
53
54     return EXIT_SUCCESS;
55 }
```

---

**Listing 9.22** – Ma version du programme créant dynamiquement un tableau de struct comme défini dans la consigne ci-dessus et le désallouant

**Question :** Testez votre programme avec valgrind pour assurer notamment qu'il n'y a pas de fuite mémoire.

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex11$ valgrind ./ex11_4
==33411== Memcheck, a memory error detector
==33411== Copyright (C) 2002–2017, and GNU GPL'd, by Julian Seward et al.
==33411== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
      info
==33411== Command: ./ex11_4
==33411==
==33411== HEAP SUMMARY:
==33411==     in use at exit: 0 bytes in 0 blocks
==33411==   total heap usage: 16 allocs, 16 frees, 1,020 bytes allocated
==33411==
==33411== All heap blocks were freed — no leaks are possible
==33411==
==33411== For lists of detected and suppressed errors, rerun with: -s
==33411== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

**Listing 9.23** – Le résultat de valgrind sur le programme ci-dessus

### 9.11.5 Tableau à deux dimensions

**Question :** Le but est de manipuler des tableaux dynamique de booléens à deux dimensions (i.e. on accède à une case avec la notation tab[3][7]).

On suppose que le premier indice correspond à la coordonnée en x et le second indice à la coordonnée en y. Si la case est à true alors le point est allumé.

Écrivez une fonction qui crée un tel tableau. Les deux paramètres sont la taille en x et la taille en y. Vous initialisez les cases comme vous le souhaitez (une fonction vous est fournie).

Écrivez une fonction qui affiche le tableau en mode pseudo-graphique, par exemple une '\*' si la case est à vrai, et '-' sinon.

Écrivez une fonction qui libère un tel tableau.

Écrivez un main qui teste vos fonctions. Par exemple la taille en x à 70 et la taille en y à 20.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4
5 #define NX 70
6 #define NY 20
7
8 typedef struct {
9     int d, f;
10 } Couple;
11
12 const Couple dessin [NX][NY+1] =
13 {
14     {{-1,-1}},
15     {{ 1, 1},{17,17},{-1,-1}},
16     {{ 1, 1},{17,17},{-1,-1}},
17     {{ 1, 1},{12,12},{18,18},{-1,-1}},
18     {{ 1, 4},{11,11},{18,18},{-1,-1}},
19     {{ 4, 4},{10,10},{18,18},{-1,-1}},
20     {{ 3, 3},{ 9, 9},{18,18},{-1,-1}},
21     {{ 2, 2},{ 8, 8},{18,18},{-1,-1}},
22     {{ 1, 1},{ 9, 9},{18,18},{-1,-1}},
23     {{10,10},{18,18},{-1,-1}},
24     {{11,11},{18,18},{-1,-1}},
25     {{12,12},{18,18},{-1,-1}},
26     {{17,17},{-1,-1}},
27     {{17,17},{-1,-1}},
28     {{ 8,12},{18,18},{-1,-1}},
29     {{17,17},{-1,-1}},
30     {{17,17},{-1,-1}},
31     {{12,12},{18,18},{-1,-1}},
32     {{11,11},{18,18},{-1,-1}},
33     {{10,10},{18,18},{-1,-1}},
34     {{ 9, 9},{18,18},{-1,-1}},
35     {{ 8, 8},{18,18},{-1,-1}},
36     {{ 9, 9},{18,18},{-1,-1}},
37     {{10,10},{18,18},{-1,-1}},
38     {{11,11},{18,18},{-1,-1}},
39     {{12,12},{18,18},{-1,-1}},
40     {{17,17},{-1,-1}},
41     {{17,17},{-1,-1}},
42     {{ 8,12},{18,18},{-1,-1}},
43     {{ 8, 8},{10,10},{12,12},{18,18},{-1,-1}},
44     {{ 8, 8},{12,12},{18,18},{-1,-1}},
45     {{ 8, 8},{12,12},{18,18},{-1,-1}},
46     {{17,17},{-1,-1}},
47     {{17,17},{-1,-1}},
48     {{17,17},{-1,-1}},
49     {{17,17},{-1,-1}},
50     {{17,17},{-1,-1}},
51     {{17,17},{-1,-1}},
52     {{17,17},{-1,-1}},
53     {{ 8,12},{18,18},{-1,-1}},
54     {{ 8, 8},{18,18},{-1,-1}},
55     {{ 8, 8},{18,18},{-1,-1}},

```

```

56    {{ 8 , 8 },{18,18},{-1,-1}},  

57    {{17,17},{-1,-1}},  

58    {{17,17},{-1,-1}},  

59    {{ 8,12 },{18,18},{-1,-1}},  

60    {{ 8 , 8 },{10,10},{12,12},{18,18},{-1,-1}},  

61    {{ 8 , 8 },{12,12},{18,18},{-1,-1}},  

62    {{ 8 , 8 },{12,12},{18,18},{-1,-1}},  

63    {{17,17},{-1,-1}},  

64    {{17,17},{-1,-1}},  

65    {{17,17},{-1,-1}},  

66    {{17,17},{-1,-1}},  

67    {{17,17},{-1,-1}},  

68    {{17,17},{-1,-1}},  

69    {{17,17},{-1,-1}},  

70    {{ 8,12 },{18,18},{-1,-1}},  

71    {{ 8 , 8 },{12,12},{18,18},{-1,-1}},  

72    {{ 8 , 8 },{12,12},{18,18},{-1,-1}},  

73    {{ 8 , 8 },{12,12},{18,18},{-1,-1}},  

74    {{17,17},{-1,-1}},  

75    {{ 1 , 1 },{17,17},{-1,-1}},  

76    {{ 2 , 2 },{17,17},{-1,-1}},  

77    {{ 3 , 3 },{17,17},{-1,-1}},  

78    {{ 4 , 4 },{17,17},{-1,-1}},  

79    {{ 1 , 4 },{17,17},{-1,-1}},  

80    {{ 1 , 1 },{17,17},{-1,-1}},  

81    {{ 1 , 1 },{ 8 , 8 },{10,12},{18,18},{-1,-1}},  

82    {{ 1 , 1 },{17,17},{-1,-1}},  

83    {{-1,-1}}  

84};  

85  

86 bool isOn(int x, int y)  

87 {  

88     if ((x < 0) || (x >= NX) || (y < 0) || (y >= NY))  

89         return false;  

90  

91     bool ok = false;  

92     int i = 0;  

93  

94     while ((! ok) && (dessin[x][i].d != -1))  

95     {  

96         if ((dessin[x][i].d <= y) && (y <= dessin[x][i].f))  

97             ok = true;  

98         i++;  

99     }  

100  

101    return ok;  

102}  

103  

104 // Début du code à écrire à partir d'ici  

105  

106 // fonction creer  

107 // la fonction isOn permet de décider si une case est à true ou false  

108 bool **create_tab(int x_size, int y_size)  

109 {  

110     bool **res = (bool **) malloc(x_size * sizeof(bool));  

111

```

```

112     for (int i = 0; i < x_size; i++)
113     {
114         res[i] = (bool *) malloc(y_size * sizeof(bool));
115     }
116
117     for (int i = 0; i < x_size; i++)
118     {
119         for (int j = 0; j < y_size; j++)
120             res[i][j] = isOn(i, j);
121     }
122
123     return res;
124 }
125
126
127 // fonction affiche
128 void print_tab(bool *tab[], int x_size, int y_size)
129 {
130     for (int j = y_size; j >= 0; j--)
131     {
132         for (int i = 0; i < x_size; i++)
133         {
134             if (tab[i][j])
135                 printf("*");
136
137             else
138                 printf("-");
139         }
140
141         printf("\n");
142     }
143 }
144
145
146 // fonction libere
147 void free_tab(bool *tab[], int x_size)
148 {
149     for (int i = 0; i < x_size; i++)
150     {
151         free(tab[i]);
152     }
153
154     free(tab);
155 }
156
157
158 int main()
159 {
160     // créer un tableau de 70 par 20
161     // afficher le tableau
162     // libérer le tableau
163     bool **mytab = create_tab(NX, NY);
164     print_tab(mytab, NX, NY);
165     free_tab(mytab, NX);
166
167     return EXIT_SUCCESS;

```

**Listing 9.24** – Le programme du tableau à double dimensions alloué et désalloué dynamiquement

**Question :** De nouveau testez votre programme avec valgrind.

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP1/Ex11$ valgrind ./ex11_5
==85182== Memcheck, a memory error detector
==85182== Copyright (C) 2002–2017, and GNU GPL'd, by Julian Seward et al.
==85182== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
==85182== info
==85182== Command: ./ex11_5
==85182==
==85182==
==85182== HEAP SUMMARY:
==85182==     in use at exit: 0 bytes in 0 blocks
==85182==   total heap usage: 72 allocs, 72 frees, 7,184 bytes allocated
==85182==
==85182== All heap blocks were freed — no leaks are possible
==85182==
==85182== For lists of detected and suppressed errors, rerun with: -s
==85182== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

**Listing 9.25** – Le résultat dans bash avec valgrind

### 9.11.6 Affectation de tableaux ?

**Question :** On ne peut pas faire une affectation de tableau (i.e. une recopie d'un tableau dans un autre avec un '=') en C. Et pourtant on peut écrire le code suivant (qui comporte par ailleurs une erreur) :

```
1 int tab[5] = {1, 2, 3, 4, 5};  
2 int * autreTab = malloc(5 * sizeof(int));  
3 autreTab = tab;
```

**Listing 9.26** – Code avec erreur

**Question :** Expliquez cette apparente contradiction, et l'erreur induite par ce code.

**Réponse :** *tab est l'adresse physique de la première case du tableau tab. autreTab est un variable dont la valeur est l'adresse du début du tableau autreTab. Le tableau tab n'est donc pas recopié dans autre tab mais il peut être accédé par tab et autreTab !*

## Exercice 12 : Tableaux

**Question :** Il n'y a pas d'allocation dynamique dans cet exercice, et on manipule des tableaux d'entiers.

Écrivez les fonctions suivantes :

- initialisation d'un tableau avec des valeurs aléatoires comprises entre deux bornes (avec rand et srand 5 )
- affichage joli d'un tableau
- moyenne des cases d'un tableau
- somme de deux tableaux au sens géométrique du terme : le résultat est un tableau dont chaque case est la somme des deux cases correspondantes des deux tableaux à sommer
- trouver le numéro de la case qui contient le second maximum du tableau (on écrit l'algorithme en supposant que toutes les cases sont distinctes)
- tri à bulle

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <time.h>
4
5 #define TAB_SIZE 100
6 #define TAB_MIN 0
7 #define TAB_MAX 101
8
9
10 int* create_tab(int tab_size)
11 {
12     int *res = (int *) malloc(tab_size * sizeof(int));
13
14     for(int i = 0; i < tab_size; i++)
15         res[i] = rand() % TAB_MAX;
16
17     return res;
18 }
19
20 void print_tab(int *res, int tab_size)
21 {
22     printf("\n");
23
24     int counter = 0;
25
26     for(int i = 0; i < tab_size; i++)
27     {
28         if(counter == 10)
29         {
30             printf("\n");
31             counter = 0;
32         }
33
34         printf("%3i ", res[i]);
35
36         counter++;
37     }
38 }
```

```

39     printf ("\n");
40 }
41
42 float avg_tab(int *res, int tab_size)
43 {
44     float sum = 0;
45
46     for (int i = 0; i < tab_size; i++)
47     {
48         sum += (float) res[i];
49     }
50
51     float result = 0;
52     result = sum / (float) tab_size;
53
54     return result;
55 }
56
57
58 int* sum_tab(int *tab1, int *tab2, int tab_size)
59 {
60     int *res = (int *) malloc(tab_size * sizeof(int));
61
62     for (int i = 0; i < tab_size; i++)
63         res[i] = tab1[i] + tab2[i];
64
65     return res;
66 }
67
68
69 int max_tab(int *tab, int tab_size)
70 {
71     int max = TAB_MIN;
72
73     for (int i = 0; i < tab_size; i++)
74     {
75         if (tab[i] > max)
76             max = tab[i];
77     }
78
79     return max;
80 }
81
82
83 int second_max_tab(int *tab, int tab_size)
84 {
85     int max = max_tab(tab, tab_size);
86     int second_max = TAB_MIN;
87
88     for (int i = 0; i < tab_size; i++)
89     {
90         if (tab[i] > second_max && tab[i] != max)
91             second_max = tab[i];
92     }
93
94     return second_max;

```

```

95 }
96
97 int* bubble_sort(int* tab, int tab_size)
98 {
99     int temp = 0;
100
101    for (int i = 0; i < tab_size - 1; i++)
102    {
103        for (int j = 0; j < tab_size - i - 1; j++)
104        {
105            if (tab[j] > tab[j+1])
106            {
107                temp = tab[j];
108                tab[j] = tab[j+1];
109                tab[j+1] = temp;
110            }
111        }
112    }
113
114    return tab;
115 }
116
117
118 int main()
119 {
120     srand(time(NULL));
121
122     int *mytab = create_tab(TAB_SIZE);
123     print_tab(mytab, TAB_SIZE);
124     float avg = avg_tab(mytab, TAB_SIZE);
125     printf("\n Average is : %f\n\n", avg);
126
127     int *mytab2 = create_tab(TAB_SIZE);
128     print_tab(mytab2, TAB_SIZE);
129
130     int *sum_tabs_res = sum_tab(mytab, mytab2, TAB_SIZE);
131     print_tab(sum_tabs_res, TAB_SIZE);
132
133     int second_max = second_max_tab(sum_tabs_res, TAB_SIZE);
134     sum_tabs_res = bubble_sort(sum_tabs_res, TAB_SIZE);
135     print_tab(sum_tabs_res, TAB_SIZE);
136     printf("Second max of last printed array is : %i\n", second_max);
137
138     free(mytab);
139     free(mytab2);
140     free(sum_tabs_res);
141
142     printf("\n");
143     return EXIT_SUCCESS;
144 }
```

**Listing 9.27** – Programme d'exercice à la manipulation des tableaux en C

# Chapitre 10

## Les chaînes de caractère en C

### Exercice 1 : Manipulations Standard

Il s'agit de manipuler les chaînes de caractères en C. Il y a une fonction par question.

Les endroits où l'on attend une réponse sont signalés par un : // ===== TODO =====

Note : Une chaîne de caractères n'est a priori qu'un simple tableau de caractères, mais avec une subtilité : la chaîne se termine par une case supplémentaire contenant le caractère de code ASCII 0 que l'on note '\0'.

note : bien que techniquement 0 et '\0' soient la même chose, ils ne le sont conceptuellement pas ; un entier et un caractère ne sont pas interchangeables

Note : un peu de vocabulaire

- un tableau de caractères contenant le mot "chat" est un tableau de 4 cases contenant 'c', 'h', 'a' et 't' C'est d'ailleurs un abus de langage de dire qu'un tableau contient un mot.
- une chaîne de caractères contenant le mot "chat" est un tableau de 5 cases contenant 'c', 'h', 'a', 't' et '\0' la longueur de la chaîne est 4 et la taille du tableau 5
- bref la difficulté est de penser à gérer cette case supplémentaire sous peine de débordement de tableau

Note : on ne fait pas d'allocation dynamique dans cet exercice

#### 10.1.1 Exercice 1a : Différence entre caractère, chaîne de caractères mutable ou non

Une variable mutable est une variable qu'on peut modifier. Donc une variable non mutable est une constante c'est un exercice très technique et propre au C

note : lorsqu'on utilise un pointeur il y a deux entités :

- le pointeur
  - la zone mémoire pointée
- Chaque entité (ou les deux) peut être constante ou non

**Question :**

```
1 // voici une série de déclarations correctes
2 char c = 'A';
3 char *s1 = "chien";
4 char s2 [] = "chiot";
5 char t [] = { 'c', 'h', 'o', 'u', 'x' };
6
```

```
7 // Afficher avec printf les 4 variables , chacune avec un code
8 // séparé
9 // ===== TODO =====
10 printf( "%c\n" , c );
11 printf( "%s\n" ,s1 );
12 printf( "%s\n" ,s2 );
13 printf( "%s\n" ,t );
```

**Résultat :**

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP2/CODE_FOURNI$ ./  
ex1 a  
=====  
= Exercice a =  
=====  
A  
chien  
chiot  
chouxchiot
```

**Réponse :** On constate donc que la variable 't' qui est un tableau de caractères sans le caractère '\0' ne s'arrête pas tant qu'il ne rencontre pas ce dernier caractère. Heureusement, "chiot" est stocké pas loin qui contient ce caractère et donc le printf s'arrête à la fin de chiot...

**Question :**

```
1 // La fonction strlen permet de récupérer la longueur d'une chaîne.  
2 // Si cela a un sens, utiliser cette fonction sur c, s1, s2 et t  
3 // et expliquer les comportements  
4 // ===== TODO =====  
5  
6 // printf("strlen(c): %lu\n", strlen(c)); //Génère Segmentation fault  
7 printf("strlen(s1): %lu\n", strlen(s1));  
8 printf("strlen(s2): %lu\n", strlen(s2));  
9 printf("strlen(t): %lu\n", strlen(t));
```

**Résultat :**

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP2/CODE_FOURNI$ ./  
ex1 a  
=====  
= Exercice a =  
=====  
strlen(s1): 5  
strlen(s2): 5  
strlen(t): 10
```

**Réponse :** On constate que la chaîne t, ne s'arrêtant pas avant celle de s2 ("chiot"), le comptage de la longueur de la chaîne 't' ne s'arrête aussi que quand il rencontre le caractère '\0'... Ce caractère est donc très important pour éviter de générer des résultats erronés !

**Question :** Quelle est la différence en s1 et s2 ?

**Réponse :** s1 est un pointeur vers un caractère mutable.

s1 a pour donnée l'adresse de son premier caractère qui est 'c'. Lors de l'initialisation un espace mémoire est alloué à "chien" en comprenant le caractère de fin de chaîne '\0' et l'adresse du début de chaîne est donné à s1.

s2 est un tableau de caractères. La donnée associée à s2 est directement le premier caractère de la chaîne.

**Question :** Modifier la 4me lettre de s1 (le 'e') et afficher la nouvelle chaîne ainsi que sa longueur. Cela marche-t-il ? La réponse est non, pourquoi ?

**Réponse :** s1[3] = 'o' génère une erreur de segment... Il semblerait donc que le segment (zone mémoire) où est stockée la chaîne de s soit en lecture seule.

**Question :** Modifier la 4me lettre de s2 (le 'o') et afficher la nouvelle chaîne ainsi que sa longueur. Cela marche-t-il ? La réponse est oui, pourquoi ?

```
1 s2[3] = 'n';
2 printf("%s\n", s2);
3 printf("%lu\n", strlen(s2));
```

**Réponse :** Cette fois-ci la chaîne de caractères est stockée dans une zone où on peut écrire (probablement la pile vu qu'il s'agit d'une variable à portée locale...)

**Question :**

```
1 // Combien de cases (si cela a un sens) ont les 4 variables
2 // ===== TODO =====
3 printf("\nMemory sizes of c, s1, s2 and t ... \n");
4 printf("sizeof(c) : %lu octet(s)\n", sizeof(c));
5 printf("sizeof(s1) : %lu octet(s)\n", sizeof(s1));
6 printf("sizeof(s2) : %lu octet(s)\n", sizeof(s2));
7 printf("sizeof(t) : %lu octet(s)\n", sizeof(t));
```

**Réponse :**

Ma réponse :

- Le contenu de c est un caractère qui fait 1 octet.
- Le contenu de s1 est un pointeur qui fait 8 octets.
- Le contenu de s2 est un tableau de caractères, il y a 6 caractères (c, h, i, o, t, 0) donc 6 octets.
- Le contenu de t est un tableau de 5 caractères donc 5 octets.

**Question :** Remplacer la 3me lettre (le i) de s2 par le caractère NUL (i.e '\0'). Afficher le nouveau contenant ainsi que la longueur. Combien de cases a le tableau s2 ?

**Réponse :**

```
1 s2[2] = '\0';
2 printf("\nReplacing third letter of s2 by '\0'...\n");
3 printf("%s\n", s2);
4 printf("%lu\n", strlen(s2));
```

**Question :** Peut-on appeler strlen sur une chaîne constante ? C'est à dire : int l = strlen("champ") ; Si oui quel est le résultat ?

**Réponse :**

```
1 int l = strlen("champ");
2 printf("\nTesting strlen() function on constant string \"champ\" \n");
3 printf("%i\n", l);
4 printf("\n");
5
6 /* Ma réponse:
7  On a donc la longueur de la chaîne sans le '\0' inclu dans le
8  compte.
9 */
```

### 10.1.2 Exercice 1b : Affectation de chaînes

Question :

```
1 const char *s1 = "chien";
2 char s2 [] = "chiot";
3
4 // le but est de recopier une chaîne dans une autre
5 // on veut donc faire des affectations (les deux lignes
6 // ci-dessus sont des initialisations et non des affectations)
7
8 // cas s1
9 // l'instruction suivante est-elle correcte et pourquoi ? Si oui
10 // que fait-elle ? (la réponse est oui)
11 printf("s1 avant : %s\n", s1);
12 s1 = "cheval";
13 printf("s1 après : %s\n", s1);
```

**Réponse :** L'instruction est correcte car lorsqu'on initialise s1 l'adresse de l'endroit où est stocké "chien" est donné à s1. Le même comportement se produit à l'affectation. On ne peut donc certes pas réécrire dans s1 comme vu dans l'exercice 1a mais on peut lui affecter une nouvelle valeur...

Question : L'instruction suivante est-elle correcte et pourquoi ? (la réponse est non)

```
1 s2 = "rat";
```

Que faut-il faire pour changer la valeur du tableau ?

**Réponse :** L'instruction suivante n'est pas correcte car "rat" génère un pointeur vers un emplacement et on essaye d'affecter s2 à ce pointeur, s2 qui est l'adresse physique vers la première case du tableau qui contient un caractère. Ce n'est donc pas possible. Par contre on peut utiliser la fonction strcpy(char \*dest, char \*src) de la bibliothèque string.h

Question : l'instruction suivante est-elle correcte et pourquoi ? (la réponse est toujours non pour les mêmes raisons)

```
1 s2 = "cheval";
```

Pourquoi ne peut-on changer la valeur du tableau pour y mettre ce mot ?

**Réponse :** cf. Explication ci-dessus...

### 10.1.3 Exercice 1c : Palindrome ?

**Question :** Écrire ici une fonction "isPalin" qui indique si la chaîne passée en paramètre est un palindrome (i.e. si la chaîne est identique qu'on la lise de gauche à droite ou de droite à gauche)

**Réponse :**

```
1 bool isPalin( const char *str )
2 {
3     int x = 0;
4     int y = strlen(str) - 1;
5
6     while(x < y)
7     {
8         if( str[x] < 'a' || str[x] > 'z' )
9             x++;
10
11        else if( str[y] < 'a' || str[y] > 'z' )
12            y--;
13
14        else if( str[x] != str[y] )
15            return false;
16
17        else
18        {
19            x++;
20            y--;
21        }
22    }
23
24    return true;
25 }
```

**Question :** Écrire ici une fonction "isPalinTab" qui indique si le tableau de caractères passé en paramètre est un palindrome. Rappel : un tableau de caractères n'est pas terminé par '\0'

**Réponse :**

```
1 bool isPalinTab( const char str[], int str_len )
2 {
3     int x = 0;
4     int y = str_len - 1;
5
6     while(x < y)
7     {
8         if(str[x] < 'a' || str[x] > 'z')
9             x++;
10
11        else if(str[y] < 'a' || str[y] > 'z')
12            y--;
13
14        else if(str[x] != str[y])
15            return false;
16
17        else
18        {
19            x++;
20            y--;
21        }
22    }
23
24    return true;
25 }
```

#### 10.1.4 Exercice 1d : Retournement d'une chaîne

Question : Ecrire ici une fonction mirror qui transforme la chaîne passée en paramètre en son miroir

Réponse :

```
1 void swap_chars(char *str, int index1, int index2)
2 {
3     char temp = str[index1];
4     str[index1] = str[index2];
5     str[index2] = temp;
6 }
7
8 void strrev(char *str)
9 {
10    int x = 0;
11    int y = strlen(str)-1;
12
13    while(x < y)
14    {
15        swap_chars(str, x, y);
16        x++;
17        y--;
18    }
19 }
```

### 10.1.5 Exercice 1e : Version personnelle de strlen

**Question :** Ecrire ici une fonction "mystrlen" qui calcule la longueur d'une chaîne (autrement dit qui fait la même chose que strlen)

**Réponse :**

```
1 int mystrlen( const char* str )
2 {
3     int i = 0;
4
5     while( str[ i ] != '\0' )
6     {
7         i++;
8     }
9
10    return i;
11 }
```

### 10.1.6 Exercice 1f : Version personnelle de strcpy

**Question :** Ecrire ici une fonction mystrcpy qui fait la même chose que strcpy.

Note : on suppose toujours, dans ce genre de fonction, que le tableau qui reçoit le résultat est déjà alloué et a une taille suffisante.

**Réponse :**

```
1 char* mystrcpy( char* dest , const char* src )
2 {
3     int i = 0;
4     for( ; src[i] != '\0'; i++)
5     {
6         dest[i] = src[i];
7     }
8     dest[i] = '\0';
9
10    return dest;
11}
12}
```

**Question :** Lorsqu'on copie une chaîne dans un autre tableau, il faut s'assurer que le tableau cible contient suffisamment de cases. D'ailleurs combien en faut-il au minimum ?

**Réponse :** Il faut "longueur de chaîne + 1" case afin de prendre en compte le caractère de terminaison de chaîne '\0'

**Question :** Que se passe-t-il s'il y en moins ?

**Réponse :** On aura un buffer overflow car on tentera alors d'écrire le caractère '\0' dans un tableau qui n'a pas suffisamment de place pour l'écrire...

**Question :** Et s'il y en a plus ?

**Réponse :** S'il y en a plus le caractère '\0' pourra être écrit et les fonctions telles que nous les écrivons s'arrêteront à ce caractère. Mais de la place mémoire sera gâchée et contiendra probablement des valeurs poubelles...

**Remarque de l'enseignant :** Prof : En l'occurrence, les tableaux ciblés sont a priori sur-dimensionnés, ce qui n'est pas une solution valide ; nous verrons des solutions correctes avec l'allocation dynamique

**Questions et réponses :**

```
1 // recopiez la source dans cible1 avec strcpy
2 // ===== TODO =====
3 strcpy(cible1, source);
4
5 // recopiez la source dans cible2 avec mystrcpy
6 // ===== TODO =====
7 mystrcpy(cible2, source);
8
9 // afficher les deux résultats avec leurs longueurs
10 // ===== TODO =====
11 printf("\nsource: %s, len: %ld\n", source, strlen(source));
12 printf("cible1: %s, len: %ld\n", cible1, strlen(cible1));
13 printf("cible2: %s, len: %ld\n", cible2, strlen(cible2));
```

### 10.1.7 Exercice 1g : strncpy (utilisation)

Question : Au fait quelle est la différence entre strlen(source) et strlen("source") ?

Réponse : strlen(source) => On demande la longueur de la chaîne contenue dans la variable source qui est un tableau de caractères contenant "bonjour". On aura donc la valeur 7, car il y a 7 caractères à "bonjour" strlen("source") => On demande la longueur de la chaîne "source", on aura donc la valeur 6 car il y a 6 caractères à "source"

Question :

```
1  char cible1 [LONGUEUR - 1];
2  char cible2 [LONGUEUR];
3  char cible3 [LONGUEUR + 1];
4  char cible4 [LONGUEUR + 2];
5
6  exo_1g_init(cible1 , LONGUEUR - 1, 'a');
7  exo_1g_init(cible2 , LONGUEUR, 'b');
8  exo_1g_init(cible3 , LONGUEUR + 1, 'c');
9  exo_1g_init(cible4 , LONGUEUR + 2, 'd');
10
11 // appeler strncpy sur source et cible1 avec LONGUEUR-1 en paramètre
12 // et afficher cible1 avec sa longueur
13 // vérifier si tout est correct et expliquer
14 // ===== TODO =====
15 strncpy(cible1 , source , LONGUEUR-1);
16 printf("\n%s : %lu\n", cible1 , strlen(cible1));
```

Réponse : strncpy permet de copier les n premiers octets (donc caractères) de la chaîne passée en source dans la chaîne de destination. Si la longueur 'n' passée en paramètre est plus grande que la chaîne source, strncpy écrit des '\0' pour être sûr que n octets sont écrits... En revanche, nous devrions avoir un problème dans l'exemple ci-dessus car aucun '\0' n'est, de ma compréhension, écrit... Le printf devrait se poursuivre jusqu'à rencontrer un null byte (ie. '\0'). Nous avons peut-être eu de la chance, le null byte ne devait se trouver pas très loin...

Questions et Réponses :

```
1  // idem avec avec cible2 et LONGUEUR
2  // ===== TODO =====
3  strncpy(cible2 , source , LONGUEUR);
4  printf("\n%s : %lu\n", cible2 , strlen(cible2));
5
6  // idem avec avec cible3 et LONGUEUR+1
7  // ===== TODO =====
8  strncpy(cible3 , source , LONGUEUR+1);
9  printf("\n%s : %lu\n", cible3 , strlen(cible3));
10
11 // idem avec avec cible4 et LONGUEUR+2
12 // ===== TODO =====
13 strncpy(cible4 , source , LONGUEUR+2);
14 printf("\n%s : %lu\n", cible4 , strlen(cible4));
```

### 10.1.8 Exercice 1h : strcmp (utilisation)

La fonction strcmp fait bien plus qu'indiquer si deux chaînes sont égales, elle indique également laquelle est la plus "petite" dans l'ordre lexicographique.

**Question :** appeler strcmp sur essais[i][0] et essais[i][1] et afficher le résultat (en indiquant également "plus petit" "égale" ou "plus grand")

```
1  char *essais [] [ 2 ] =
2  {
3      {"chat", "chat"}, 
4      {"chat", "chien"}, 
5      {"chien", "chat"}, 
6      {"chat", "chaton"}, 
7      {"souris", "elephant"}, 
8      {"souris", "éléphant"}, 
9      {"anticonstitutionnellement", "zoo"}, 
10     {"145", "237"}, 
11     {"1453", "22"}, 
12     {"+23", "-23"} 
13 }
```

Réponse :

```
1  for ( int i = 0; i < nbCouples; i++)
2  {
3      // appeler strcmp sur essais[i][0] et essais[i][1]
4      // et afficher le résultat (en indiquant également "plus petit" "é
5      // gale"
6      // ou "plus grand")
7      // ===== TODO =====
8
9
10     int comparison_result = strcmp( essais [ i ][ 0 ] , essais [ i ][ 1 ] );
11
12     if (comparison_result < 0)
13         printf( "%s plus petit que %s\n" , essais [ i ][ 0 ] , essais [ i ][ 1 ] );
14
15     else if (comparison_result == 0)
16         printf( "%s égal à %s\n" , essais [ i ][ 0 ] , essais [ i ][ 1 ] );
17
18     else
19         printf( "%s plus grand que %s\n" , essais [ i ][ 0 ] , essais [ i ][ 1 ] );
20 }
```

### 10.1.9 Exercice 1i : strcmp (version personnelle)

**Question :** Ecrire ici une fonction mystrcmp équivalente à strcmp proposez un code optimisé, même si c'est au détriment de la lisibilité

**Réponse :**

```
1 int mystrcmp( const char* s1 , const char* s2 )
2 {
3     unsigned char c1;
4     unsigned char c2;
5     for( int i = 0; s1[ i ] != '\0' || s2[ i ] != '\0'; i++)
6     {
7         c1 = ( unsigned char ) s1[ i ];
8         c2 = ( unsigned char ) s2[ i ];
9
10        //printf("%c vs %c\n", s1[ i ], s2[ i ]);
11        //printf("%i vs %i\n", s1[ i ], s2[ i ]);
12
13        if( c1 != c2 )
14            return c1 - c2;
15    }
16
17    return c1 - c2;
18 }
```

### 10.1.10 Exercice 1j : strcat (utilisation)

**Question :** La fonction strcat permet de contacténer une chaîne à une autre. La difficulté est de s'assurer que le tableau contenant la chaîne à allonger est suffisamment grand.

**Réponse :**

```
1 void exo_1j()
2 {
3     // on surdimensionne les tableaux pour éviter la gestion mémoire
4     char s1[1000] = "il fait beau";
5     char s2[1000] = " et chaud";
6
7     // afficher s1 et s2 et leurs longueurs
8     // ===== TODO =====
9     printf("%s : %lu\n", s1, strlen(s1));
10    printf("%s : %lu\n", s2, strlen(s2));
11
12    // concaténer s2 à la fin de s1
13    // ===== TODO =====
14    strcat(s1, s2);
15
16    // afficher s1 et sa longueur
17    // ===== TODO =====
18    printf("%s : %lu\n", s1, strlen(s1));
19 }
```

### 10.1.11 Exercice 1k : strstr (utilisation)

La fonction strstr permet de rechercher une sous-chaîne dans une chaîne

Questions et Réponses :

```
1 // décommenter une seule ligne
2 const char * s[2] = { "Il fait beau et chaud", "beau" };
3 //const char * s[2] = { "Il fait beau et chaud", "i" };
4 //const char * s[2] = { "Il fait beau et chaud", "au" };
5 //const char * s[2] = { "Il fait beau et chaud", "au " };
6 //const char * s[2] = { "Il fait beau et chaud", "aud" };
7 //const char * s[2] = { "Il fait beau et chaud", "froid" };
8 //const char * s[2] = { "Il fait beau et chaud", "x" };
9
10 // Indiquer si s[1] est une sous-chaîne de s[0]
11 // ===== TODO =====
12 const char *substr = strstr(s[0], s[1]);
13 printf("\n%s\n", substr);
```

**Question :** Plus compliqué : si s[1] est une sous-chaîne de s[0], indiquer à quelle position se trouve cette sous-chaîne

Réponse :

```
1 void findPosV2(const char *str, const char *substr)
2 {
3     if(strstr(str, substr) != NULL)
4     {
5         int res = 0;
6         int j = 0;
7         bool checking = false;
8
9         for(int i = 0; str[i] != '\0'; i++)
10        {
11            if(substr[0] == str[i] && !checking)
12            {
13                res = i;
14                j++;
15                checking = true;
16            }
17
18            else if(substr[j] == '\0')
19                break;
20
21            else if(substr[j] == str[i] && checking)
22            {
23                j++;
24            }
25
26            else
27            {
28                j = 0;
29                checking = false;
30            }
31        }
32    }
```

```
32         fprintf(stdout , "substring is at position: %i\n" , res);
33     }
34
35     else
36         fprintf(stderr , "Provided substring isn't in string , can't proceed\n");
37     }
38
39
40
41 int findPos(const char* str , const char* substr)
42 {
43     if(substr != NULL)
44         return substr - str;
45
46     else
47         return -1;
48 }
```

### 10.1.12 Exercice 11 : sprintf ou commenter formater une chaîne

Cette fonction est exactement la même que printf, si ce n'est qu'au lieu d'être affiché à l'écran, le résultat est mis dans une chaîne

Questions et Réponses :

```
1 const int MAX = 10000;
2 char s[MAX];
3
4 int n = 5;
// mettre dans s la phrase : "Il y a <n> pingouins" où <n> doit être
// remplacé par la valeur de n
// ===== TODO =====
5
6
7
8
9 sprintf(s, "Il y a %i pingouins", n);
10 printf("vérif : %s\n", s);
```

Questions et Réponses :

```
1 // mettre dans s la table de multiplication jusqu'à n
2 // par exemple si n contient 4, s contiendra (retour chariot compris) :
3 //      n  1  2  3  4
4 //      1  1  2  3  4
5 //      2  2  4  6  8
6 //      3  3  6  9 12
7 //      4  4  8 12 16
8 // ===== TODO =====
9
10 char temp[5];
11 sprintf(s, "\n");
12
13 for(int i = 0; i < n+1; i++)
14 {
15     for(int j = 0; j < n+2; j++)
16     {
17         if(i == 0 && j == 0)
18             strcat(s, "\n    ");
19
20         else if(j == n+1)
21         {
22             sprintf(temp, " \n ");
23             strcat(s, temp);
24         }
25
26         else if(i == 0)
27         {
28             sprintf(temp, " %2i ", j);
29             strcat(s, temp);
30         }
31
32         else if(j == 0)
33         {
34             sprintf(temp, " %2i ", i);
35             strcat(s, temp);
36         }
37 }
```

```
38     else if(j == n+1 && i == n)
39     {
40         strcat(s, "\0");
41     }
42
43     else
44     {
45         sprintf(temp, " %2i ", i*j);
46         strcat(s, temp);
47     }
48 }
49
50 printf("vérif : %s\n", s);
```

### 10.1.13 Exercice 1m : Du bon usage des fonctions

**Question :** Pourquoi ce code est incorrect, i.e. ne fonctionne pas ? On veut ajouter la lettre 'a' à la fin de la chaîne autant de fois qu'il y a de caractères dans la chaîne. ex : si  $s = "AAA"$  alors  $s$  devient " $AAAAaa$ " ex : si  $s = "azer"$  alors  $s$  devient " $azeraaaa$ "

```
1 strcpy(s, "AAA");
2 printf("\nstrlen(s) = %lu\n", strlen(s));
3
4 for (unsigned int i = 0; i < strlen(s); i++)
5 {
6     strcat(s, "a");
7 }
```

**Réponse :**

```
1 strcpy(s, "AAA");
2 printf("\nstrlen(s) = %lu\n", strlen(s));
3
4 unsigned int s_len = strlen(s);
5
6 for (unsigned int i = 0; i < s_len; i++)
7 {
8     strcat(s, "a");
9 }
```

En raisonnant comme un programme on fait donc : 1) On initialise  $i$  variable locale à 0, on entre dans la boucle, on exécute  $\text{strcat}(s, "a")$  qui réécrit par-dessus le null byte '\0' et le rerajoute en bout de chaîne.

2) On teste si la condition de la boucle est invalidée. Le calcul  $\text{strlen}(s)$  est donc effectué. La chaîne vaut 4 (alors qu'avant  $\text{strcat}$  elle valait 3 !)

3) On incrémente  $i$  et on retourne dans la boucle.

$s$  est un tableau de caractères pouvant accueillir 10 000 caractères...  $i$  s'incrémentera jusqu'à 10 000 puis le comportement devient imprédictible... Pour palier à ça on fait donc le calcul  $\text{strlen}(s)$  une fois et on stocke le résultat dans  $s\_len$  qu'on utilise ensuite dans la boucle  $for$ .

**Question :** Pourquoi ce code, bien que donnant le bon résultat, est incorrect ? On remplace tous les espaces par des points...

```
1 strcpy(s, "il fait beau ou chaud");
2 // ===== début TODO =====
3 for (unsigned int i = 0; i < strlen(s); i++)
4 {
5     if (s[i] == ' ')
6         s[i] = '.';
```

**Réponse :** Pour la même raison que ci-dessus, on refait le calcul  $\text{strlen}(s)$  à chaque tour de boucle ce qui pour la complexité en temps n'est pas une bonne idée et, si on veut réaliser d'autres opérations dans la boucle qui modifie la longueur initiale de la chaîne nous retrouverons le problème précédent...

### 10.1.14 Exercice 1n : == ou strcmp ?

Questions et Réponses :

```
1 // Pourquoi faut-il utiliser strcmp et non == pour comparer 2 chaînes ?
2 // ===== TODO =====
3
4 // illustration 1
5 char t1a[] = "chaton";
6 char *t1b = "chaton";
7 if (t1a == t1b)
8     printf("vrai");
9 else
10    printf("faux");
11 printf(" : un [] et un char *\n");
12 // Expliquez
13 // Ma réponse: Avec == on compare l'égalité de référence et non de
14 contenu
15 // ===== TODO =====
16
17 // illustration 2
18 char t2a[] = "chaton";
19 char t2b[] = "chaton";
20 if (t2a == t2b)
21     printf("vrai");
22 else
23    printf("faux");
24 printf(" : deux []\n");
25 // Expliquez
26 // Ma réponse: Même problème que précédemment, quand on crée deux
27 tableaux
28 // de caractères avec affectations de chaînes il y a deux zones mé
29 moires
30 // différentes allouées qui contiennent la chaîne "chaton".
31 // Par conséquence on a deux références différentes.
32 // ===== TODO =====
33
34 // illustration 3
35 char *t3a = "chaton";
36 char *t3b = "chaton";
37
38 if (t3a == t3b)
39     printf("vrai");
40 else
41    printf("faux");
42 printf(" : deux char *\n");
43 // Expliquez
44 // Ma réponse: Il semblerait qu'une optimisation mémoire est ici réalis
45 ée
46 // où la chaîne "chaton" étant la même, t3b pointe vers la même zone mé
47 moire
48 // que t3a. Elles ont alors la même référence et donc l'égalité est
49 vraie.
50 // ===== TODO =====
```

```

48
49 // illustration 4
50 char *t4a = "chaton";
51 char *t4b = t4a;
52 if (t4a == t4b)
53     printf("vrai");
54 else
55     printf("faux");
56 printf(" : deux char *\n");
57 // Expliquez
58 // Ma réponse: On fait pointer t4b vers le début de la chaîne contenue
59 // par t4a.
60 // On a donc deux 'accès' à la même chaîne, les références sont les mêmes donc
61 // le résultat du test est vrai.
62 // ===== TODO =====
63
64 // illustration 5
65 char t5a[] = "chaton";
66 char *t5b = t5a;
67
68 printf("\nt5a at %p contains %c\n\n", (void *) t5a, *t5a);
69
70 if (t5a == t5b)
71     printf("vrai");
72 else
73     printf("faux");
74 printf(" : un [] et un char *\n");
75 // Expliquez
76 // Ma réponse: Même remarque que pour l'illustration 4, un tableau est une
77 // adresse physique vers le début de ce qu'il contient...
78 // ===== TODO =====
79
80 // illustration 6
81 char *t6a = strdup("chaton");
82 char *t6b = strdup("chaton");
83 if (t6a == t6b)
84     printf("vrai");
85 else
86     printf("faux");
87 printf(" : deux char * avec malloc\n");
88 // Expliquez
89 // Ma réponse: strdup est une fonction spéciale qui permet d'éviter
90 // l'inconvénient vu en illustration 3. La fonction permet d'
91 // initialiser
92 // deux chaînes en deux zones mémoires différentes avec le même contenu
93 .
94 // Pour cela elle fait un malloc.
95 // ===== TODO =====
96 free(t6a);
97 free(t6b);

```

## Exercice 2 : Allocation Dynamique

### 10.2.1 Exercice 2a : Clone d'une chaîne

Le but est de dupliquer une chaîne avec allocation dynamique : la fonction alloue l'espace nécessaire avant de faire la copie.

**Question :** Écrire ici une fonction "strClone" qui prend en paramètre la chaîne à copier et renvoie la copie. On alloue la mémoire juste nécessaire. Interdiction d'utiliser les fonctions de la bibliothèque string (à part strlen)

Réponse :

```
1 char* strClone( const char* s )
2 {
3     int s_len = strlen(s);
4     char* res = (char*) malloc((s_len+1) * sizeof(char));
5
6     for (int i = 0; i < s_len; i++)
7         res[i] = s[i];
8
9     res[s_len] = '\0';
10
11    return res;
12 }
```

**Remarque personnelle :** Allocation dynamique => free() !!!

### 10.2.2 Exercice 2b : strcat version dynamique

Le but est d'écrire une version particulière de strcat : la chaîne à allonger a été allouée dynamiquement et la fonction doit la réallouer à la bonne taille.

**Question :** Écrire ici une fonction "strDynCat" qui prend en paramètre la chaîne à copier et la chaîne à ajouter. On respecte les fonctionnalités classique de strcat avec deux différences :

- donc la destination doit être réallouée.
- la fonction renvoie void

Il est autorisé d'utiliser les fonctions de la bibliothèque string. Le prototype est particulièrement délicat.

**Réponse :**

```
1 void strDynCat( char **dest_ptr, const char *src )
2 {
3     int src_len = strlen( src );
4     int dest_len = strlen(*dest_ptr);
5     int new_len = src_len + dest_len;
6
7     *dest_ptr = realloc( *dest_ptr, new_len+1 );
8
9     int j = 0;
10    for( int i = dest_len; i < new_len; i++ )
11    {
12        (*dest_ptr)[ i ] = src[ j ];
13        j++;
14    }
15
16    (*dest_ptr)[ new_len ] = '\0';
17
18 }
```

### 10.2.3 Exercice 2c : Formatage et allocation

**Question :** La contrainte est particulière : on veut formater une chaîne (avec sprintf). Mais on veut que la chaîne qui reçoit le résultat soit allouée au plus juste.

La démarche est obligatoirement celle-ci :

- on alloue la chaîne
- on appelle sprintf pour la remplir

Mais pour allouer la chaîne au mieux, il faut avoir le résultat de sprintf auparavant, et donc on est dans une impasse. Il est interdit de surdimensionner tout simplement parce que dans un cas général (que nous ne ferons pas pour simplifier) il est impossible d'avoir une idée de la taille maximale.

Voici le code que l'on NE veut PAS :

```
1 res = malloc(10000 * sizeof(char));      // on est tranquille (!! vraiment ?)
2 sprintf(res, format, r1, r2);
```

Et il faudrait ici réallouer pour raccourcir la chaîne; même si ce n'est pas le propos ici, regardez comment on fait.

Ecrivez votre propre code pour que le malloc, avant le sprintf, soit directement de la bonne taille. Pour calculer cette taille, étudiez la fonction snprintf et regardez comment elle se comporte lorsque la taille fournie n'est pas suffisante.

**Réponse :**

```
1 int format_len = snprintf(NULL, 0, format, r1, r2) + 1;
2 //+1 pour permettre l'écriture du null byte '\0'...
3
4 res = (char *) malloc(format_len * sizeof(char));
5 sprintf(res, format, r1, r2);
```

#### 10.2.4 Exercice 2d : Retour sur les chaînes littérales

Questions et Réponses :

```
1  /*const*/ char *s1 = "Tu vois , le monde se divise en deux catégories";
2  /*const*/ char *s2 = "Tu vois , le monde se divise en deux catégories";
3
4  // Affichez et comparez les valeurs des deux pointeurs.
5  // Expliquez ce qu'il se passe (attention spoiler dans la suite)
6  // ===== TODO =====
7  printf("\ns1: %p\n", (void *) s1);
8  printf("\ns2: %p\n", (void *) s2);
9  //Ma réponse: Les deux pointeurs ont la même valeur. Il se peut que,
10 //les chaînes étant constantes et identiques , le compilateur
11 //ait optimisé leurs occupations mémoires en association les deux
12 //pointeurs sur une même zone mémoire...
13
14 // On s'aperçoit que les deux pointeurs ont la même valeur.
15 // Donc que ce passe-t-il (il faudrait enlever le const) si
16 // on modifie la chaîne pointée par s1 ? Normalement la chaîne
17 // pointée par s2 est aussi modifiée ?
18 // Essayez
19 // ===== TODO =====
20 s1 = "test";
21 printf("\ns1: %p\n", (void *) s1);
22 printf("\ns2: %p\n", (void *) s2);
23
24 //Ma réponse: On ne peut pas modifier s1 même en enlevant le const
25 //car char *s1 = "ma chaine" est mise dans une zone read-only et donc
26 //traitée comme une chaîne constante par défaut. D'où le fait que j'ai
27 //écrit ci-dessus s1 = "test" , tout ce que je pouvais faire était une
28 //réaffectation à s1.
29
30 s1 = NULL;
31 s2 = NULL;
32 // la zone mémoire qui contient la chaîne n'est plus référencée.
33 // A-t-on une fuite mémoire ?
34 // ===== TODO =====
35 //
36 // Ma réponse: Il n'y a pas de fuite mémoire car s1 et s2 sont
37 // déréférencée et n'ont pas été allouée dynamiquement donc elles
38 // sont libérées. Un lancement avec valgrind semble confirmer cette
39 // hypothèse...
```

# Chapitre 11

## Modules

### Définir ou déclarer ?

#### 11.1.1 Explications

##### Point de cours

On a souvent tendance à utiliser indifféremment les verbes “définir” et “déclarer” pour une variable ou une fonction (ou une classe).

Voici les significations correctes :

- *définir* signifie “créer”
- *déclarer* signifie “être défini ailleurs”

On ne peut donc définir qu’une seule fois une entité et on peut la déclarer plusieurs fois.

Pour une fonction, c’est non ambigu :

- pour la *déclarer* on met uniquement le prototype (généralement dans un *.h*)
- pour la *définir* on indique le prototype et le corps de la fonction (généralement dans un *.c*)

Pour une variable locale (à un bloc d’instructions), il y a encore moins d’ambiguïté : une variable locale ne peut qu’être *définie*. Il est préférable, pour des raisons de lisibilité, de définir les variables locales en début de bloc (juste après l’accolade ouvrante).

Pour les variables globales c’est plus délicat car la même instruction peut être une définition ou une déclaration selon le contexte.

S’il y a une initialisation alors c’est obligatoirement une définition :

```
int nb = 3;
```

S’il n’y a pas d’initialisation alors ce peut être une déclaration ou une définition :

```
int nb;
```

Comme il est “hors de question” de laisser un code aussi peu lisible, on imposera l’utilisation du mot-clé *extern* pour les déclarations (cf. ci-dessous).

### 11.1.2 Exercice

**Question :** Le but est d'écrire une fonction, nommée afficheN, qui affiche plusieurs fois un messages (fonction à deux paramètres donc). Écrivez un fichier .c qui contient dans cet ordre :

- une déclaration de afficheN
- une déclaration de afficheN
- la définition de afficheN
- une déclaration de afficheN
- la fonction main (sa définition) qui, dans le corps, déclare la fonction afficheN et l'appelle.

L'exercice a pour seul intérêt de montrer qu'on peut déclarer plusieurs fois une entité.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // déclarer afficheN
5 // todo
6 void afficheN(char *str, int n);
7
8 // déclarer afficheN
9 // todo
10 void afficheN(char *str, int n);
11
12 // définir afficheN
13 // todo
14 void afficheN(char *str, int n)
15 {
16     for(int i = 0; i < n; i++)
17         printf("%s\n", str);
18 }
19
20 // déclarer afficheN
21 // todo
22 void afficheN(char *str, int n);
23
24 int main()
25 {
26     // déclarer afficheN
27     // todo
28     void afficheN(char *str, int n);
29
30     // appeler afficheN pour afficher 3 fois "Bonjour"
31     // todo
32     afficheN("Bonjour", 3);
33
34     return EXIT_SUCCESS;
35 }
```

**Listing 11.1** – code source du main.c de l'exercice 1.2

### 11.1.3 Exercice

**Question :** Le but est de faire de la récursivité croisée avec deux fonctions moins1 et moins2. Chacune prend un entier en paramètre, soustrait un nombre (1 ou 2 donc) et appelle l'autre fonction pour continuer les calculs. On arrête la récursivité lorsqu'on arrive à un nombre inférieur ou égal à 1.

Voici le code :

```
moins1
1 int moins1(int n) // définition
2 {
3     if (n <= 1)
4         return n;
5     n -= 1;
6     return moins2(n);
7 }

moins2
1 int moins2(int n) // définition
2 {
3     if (n <= 1)
4         return n;
5     n -= 2;
6     return moins1(n);
7 }
```

Dans un seul fichier .c (sans .h) écrivez ces deux méthodes et un main qui en appelle une. Le fichier doit compiler avec les options habituelles et sans warning.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // todo : écrire les fonctions moins1 et moins2
5 // code fourni en fin de fichier pour éviter de tout retaper
6 int moins1(int n);
7 int moins2(int n);
8
9 int main()
10 {
11     int r = moins2(9);
12     printf("%d\n", r);
13
14     return EXIT_SUCCESS;
15 }
16
17
18 int moins1(int n) // définition
19 {
20     if (n <= 1)
21         return n;
22     n -= 1;
23     return moins2(n);
24 }
25
26 int moins2(int n) // définition
27 {
28     if (n <= 1)
29         return n;
30     n -= 2;
31     return moins1(n);
32 }
```

**Listing 11.2** – code source du main.c de l'exercice 1.3

## Notion de module

### 11.2.1 Définitions

#### Point de cours

Un couple de fichiers `.h/.c` est en règle générale un module (on fera souvent l'abus de dire que c'est une classe au sens objet du terme).

C'est donc une entité :

- spécialisée : elle ne gère qu'un seul concept de manière complète, par exemple un vecteur géométrique, une image, une voiture, ...
- autonome (si possible) : on entend par ce terme qu'elle peut être réutilisée dans un autre logiciel sans changement.

Dans notre contexte il y a deux types de personnes <sup>a</sup> :

- le concepteur du module : c'est un développeur considéré comme chevronné. C'est lui qui écrit le couple `.h/.c`.
- l'utilisateur du module (ou bibliothèque) : c'est aussi un développeur <sup>b</sup>, client du module (pour simplifier celui qui écrit le fichier `main.c`).

Un concepteur partira toujours du principe que l'utilisateur est un mauvais programmeur, et donc il protégera son module au maximum contre une utilisation erronée.

Le fichier `.h` est l'interface publique qui contient uniquement les informations utiles pour l'utilisateur du module. C'est le "quoi", i.e. la description de ce que peut faire le module.

Le fichier `.c` est l'implémentation du module ; l'utilisateur du module n'a pas utilité de regarder ce fichier, et très souvent d'ailleurs il ne le peut pas physiquement (lorsque le code est fourni sous forme de bibliothèque pré-compilée). C'est le "comment", i.e. les solutions choisies et leurs implémentations.

Théoriquement le `.h` devrait être écrit avant le `.c`. C'est en fait assez logique, il est préférable de savoir ce qu'on veut faire avant de se demander comment on le fera.

Le `.c` doit pouvoir être modifié à tout moment sans impact pour les utilisateurs (si ce n'est la performance).

En revanche la modification d'un `.h` impose souvent à l'utilisateur de modifier une partie importante de son code : il faut particulièrement soigner l'écriture de ce fichier.

<sup>a</sup> de manière schizophrénique, nous jouerons les deux rôles.

<sup>b</sup> l'utilisateur final (i.e. la personne utilisant le logiciel), qui n'est a priori pas un informaticien, nous intéresse très peu dans ce cours.

### 11.2.2 Exercice

**Question :** Créez un module ImageMono qui représente une image mono-couleur (i.e. tous les pixels ont la même couleur).

Une telle image contient :

- la largeur et la hauteur (des entiers)
- la couleur (une chaîne de caractères)

Note : on ne gérera pas la mémoire ; la chaîne aura une taille en dur et on supposera que le nom d'une couleur ne sera jamais trop grand 3 .

Les opérations (i.e. les méthodes/fonctions du module) possibles sont (et la liste est exhaustive) :

- initialiser les données (largeur, hauteur, couleur) ; c'est un "constructeur"
- récupérer la largeur
- récupérer la hauteur
- récupérer la couleur
- changer la couleur

Notamment ce module interdit la modification de la largeur ou de la hauteur après l'initialisation. Le fichier main.c :

- créera une image bleu
- changera la couleur en vert
- affichera les 3 informations

Écrivez, dans cet ordre :

- ImageMono.h
- main.c que vous compilerez (option -c de gcc ou "make main.o") avant d'écrire ImageMono.c
- ImageMono.c

Testez votre programme avec valgrind.

Comparez votre solution avec celle fournie et regardez si vous êtes en accord avec les remarques.

**Réponse :**

```
1 // todo
2
3 typedef struct
4 {
5     int length, height;
6     char* color;
7 } image;
8
9 /* — initialiser les données (largeur, hauteur, couleur) ; c'est un "
   * constructeur"
10 * — récupérer la largeur
11 * — récupérer la hauteur
12 * — récupérer la couleur
13 * — changer la couleur*/
14
15 image initImage(int length, int height, char* color);
16 int getImageLen(const image* im);
17 int getImageHeight(const image* im);
18 char* getImageColor(const image* im);
19 void setImageColor(image* im, char* color);
```

**Listing 11.3** – code source de mon ImageMono.h

```

1 // todo
2 #include "ImageMono.h"
3
4 image initImage(int length, int height, char* color)
5 {
6     image res = {length, height, color};
7     return res;
8 }
9
10 int getImageLen(const image* im)
11 {
12     return im->length;
13 }
14
15 int getImageHeight(const image* im)
16 {
17     return im->height;
18 }
19
20 char* getImageColor(const image* im)
21 {
22     return im->color;
23 }
24
25 void setImageColor(image* im, char* color)
26 {
27     im->color = color;
28 }
```

**Listing 11.4** – code source de mon ImageMono.c

```

1 // todo
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include "ImageMono.h"
6
7 int main()
8 {
9     image myImage = initImage(100, 100, "blue");
10    setImageColor(&myImage, "green");
11    printf("\nImage has length %i, height %i and color %s\n",
12          getImageLen(&myImage), getImageHeight(&myImage),
13          getColor(&myImage));
14    return EXIT_SUCCESS;
15 }
```

**Listing 11.5** – code source de mon main.c

```
(base) florian@CRexJr:~/.../2_2_EXERCICE$ valgrind ./main
==69879== Memcheck, a memory error detector
==69879== Copyright (C) 2002–2017, and GNU GPL'd, by Julian Seward et al.
==69879== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
      info
==69879== Command: ./main
==69879==

Image has length 100, height 100 and color green
==69879==
==69879== HEAP SUMMARY:
==69879==   in use at exit: 0 bytes in 0 blocks
==69879==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==69879==
==69879== All heap blocks were freed — no leaks are possible
==69879==
==69879== For lists of detected and suppressed errors, rerun with: -s
==69879== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

**Listing 11.6** – Résultat de l'exécution du programme ci-dessus dans Bash et en utilisant Valgrind

# Mot-clé static

## 11.3.1 Fonctions static

### Définition

#### Point de cours

Une fonction *static* est une fonction qui n'est connue et appelleable que dans le fichier *.c* où elle est définie/déclarée. Autrement dit, une fonction *static* est locale au fichier de définition, et elle n'est pas utilisable dans un autre fichier *.c*.

Cela n'a donc pas de sens (il est même incorrect) de déclarer/définir une fonction *static* dans un *.h*.

Il y a deux intérêts aux fonctions *static* :

- Le principal est donc qu'elles sont locales à un seul fichier ; elles sont donc privées au sens objet du terme.
- On peut avoir plusieurs fonctions qui ont le même nom à condition qu'elles soient *static* et dans des fichiers *.c* différents.

Il est fortement déconseillé d'avoir une fonction *static* et une fonction *non-static* qui ont le même nom. En général cela conduit à une erreur de compilation.

Le seul cas où cela ne pose pas de problème, c'est lorsque le module est autonome, car par définition il n'appelle pas de fonctions extérieures (hormis les fonctions propres au langage).

En fait il y a une manière d'appeler une fonction *static* hors de son fichier de description, mais ce n'est pas abordé dans ce cours<sup>a</sup>.

Syntaxe :

```
static <type> <nom_fonction>([paramètres])
```

Exemple :

```
static float moyenne(const int t[], int n)
```

<sup>a</sup>. Pour les curieux, il faut récupérer un pointeur sur cette fonction, ce qui n'est possible que si le module le permet explicitement.

### Exercice

```
1 #include <stdio.h>
2
3 void f()
4 {
5     printf("Hello ");
6 }
7
8 void g()
9 {
10    printf("World!\n");
11 }
```

**Listing 11.7** – code source de fonctions.h sans fonction static

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     f();
7     g();
8 }
```

**Listing 11.8** – code source du main.c de l'exercice 3.1.2

```
(base) florian@CRegJr:~/.../3_1_2_EXERCICE$ gcc -Wall -Wextra -pedantic -
      std=c99 main.o fonctions.o -o main

(base) florian@CRegJr:~/.../3_1_2_EXERCICE$ ./main
Hello World!
```

**Listing 11.9** – Résultat de l'édition de liens avec le fonctions.h sans fonction static

```

1 #include <stdio.h>
2
3 void f()
4 {
5     printf("Hello ");
6 }
7
8 static void g()
9 {
10    printf("World!\n");
11 }
```

**Listing 11.10** – code source de fonctions.h avec fonction static

```
(base) florian@CRegJr:~/.../3_1_2_EXERCICE$ gcc -Wall -Wextra -pedantic -
      std=c99 -c fonctions.c

(base) florian@CRegJr:~/.../3_1_2_EXERCICE$ gcc -Wall -Wextra -pedantic -
      std=c99 main.o fonctions.o -o main
/usr/bin/ld: main.o: in function `main'
main.c:(.text+0x18): undefined reference to `g'
collect2: error: ld returned 1 exit status
```

**Listing 11.11** – Résultat de l'édition de liens avec le fonctions.h avec fonction static

### 11.3.2 Variable static définie hors des fonctions

#### Définition

##### Point de cours

Rappelons que sans *static*, une variable définie hors des fonctions est persistante/permanente et globale ; elle est connue de tous les *.c*.

Une telle variable *static* a le même comportement qu'une fonction *static* : elle est toujours persistante, mais locale au fichier de définition.

Elle n'est ni connue, ni accessible dans un autre *.c*.

Corolaire, si l'instruction :

```
static int v = 3;
```

est écrite dans deux *.c* différents, alors chaque fichier *.c* a sa propre variable *v* qui n'interfère pas avec l'autre fichier.

Deux variables *static* de même nom (dans des fichiers différents) peuvent avoir des types différents.

#### Exercice

```
1 int g;
2 static int s;
```

Listing 11.12 – code source de variables.c

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     extern int g;
7     extern int s;
8
9     printf("Non static: %i , Static: %i\n", g, s);
10 }
```

Listing 11.13 – code source du main.c de l'exercice 3.2.2

```
crex@crex:~/.../3_2_2_EXERCICE$ gcc -Wall -Wextra -pedantic -std=c99 -c
main.c
crex@crex:~/.../3_2_2_EXERCICE$ gcc -Wall -Wextra -pedantic -std=c99 -c
variables.c
variables.c:2:12: warning: "s" defined but not used [-Wunused-variable]
  2 | static int s;
    |
    ^

crex@crex:~/.../3_2_2_EXERCICE$ gcc -Wall -Wextra -pedantic -std=c99 main.o
variables.o -o main
/usr/bin/ld: main.o: in function 'main':
main.c:(.text+0xa): undefined reference to 's'
collect2: error: ld returned 1 exit status
```

Listing 11.14 – Résultat du préprocesseur et éditeur de liens sur les fichiers sources de l'exercice 3.2.2 mentionnés ci-dessus

### 11.3.3 Variable static locale à une fonction

#### Définition

##### Point de cours

Pour une variable locale, la notion de *static* est très différente.

Appuyons-nous sur un exemple :

```
f_correct          f_faux
1 void f_correct() 1 void f_faux()
2 {                {
3     static int n = 3; 3     static int n;
4     n++;             n = 3;
5     printf("%d\n", n); 4     n++;
6 }                 5     printf("%d\n", n);
7 }
```

Une telle variable est locale à la fonction (comme une variable normale) et n'est donc pas utilisable hors de la fonction.

En revanche elle est persistante : sa valeur est conservée entre deux appels.

On en déduit que la variable *n* n'est initialisée qu'une seule fois lors du premier appel. Autrement dit, la ligne 3 est exécutée lors du premier appel et ignorée lors des appels suivants.

C'est pour cela que la version de droite est erronée : la ligne 3 est exécutée lors du premier appel (et *n* est initialisée à 0) ; en revanche la ligne 4 est exécutée chaque fois ce qui fait perdre tout l'intérêt d'avoir une variable *static*.

```
main              main (suite)
1 int main()      7 f_faux();
2 {               8 f_faux();
3     f_correct(); 9 f_faux();
4     f_correct(); 10
5     f_correct(); 11     return EXIT_SUCCESS;
6 }
```

Les lignes 3 à 5 vont afficher successivement 4, 5 et 6 :

- lors du premier appel, *n* est initialisée à 3 et passe à 4
- Lors du deuxième appel, la première ligne de la fonction est ignorée et *n* passe à 5
- Lors du troisième appel, la première ligne de la fonction est ignorée et *n* passe à 6

Les lignes 7 à 9 vont afficher trois fois la valeur 4 :

- lors du 1<sup>er</sup> appel, *n* est initialisée à 0, passe à 3 et passe à 4
- Lors du 2<sup>me</sup> appel, la première ligne de la fonction est ignorée, *n* passe à 3 et passe à 4
- Lors du 3<sup>me</sup> appel, la première ligne de la fonction est ignorée, *n* passe à 3 et passe à 4

Ce n'est vraisemblablement pas ce qui était voulu.

## Exercice

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 void f_correct()
5 {
6     static int n = 3;
7     n++;
8     printf("%i\n", n);
9 }
10
11
12 void f_faux()
13 {
14     static int n;
15     n = 3;
16     n++;
17     printf("%i\n", n);
18 }
19
20 int main()
21 {
22     f_correct();
23     f_correct();
24     f_correct();
25
26     printf("\n");
27
28     f_faux();
29     f_faux();
30     f_faux();
31 }
```

**Listing 11.15** – code source du main.c de l'exercice 3.3.2

```
crex@crex:~/.../3_3_2_EXERCICE$ ./main
4
5
6
4
4
4
```

**Listing 11.16** – Résultat de l'exécution du programme de l'exercice 3.3.2 mentionné ci-dessus

## Mot-clé *extern*

### Point de cours

Normalement, le mot *extern* devrait être placé devant toute déclaration (et non définition) d'une variable globale ou d'une fonction globale.

Cependant il est systématiquement omis pour les fonctions. En effet il n'y a pas d'ambiguïté pour une fonction :

- soit il n'y a pas de corps et il s'agit d'une déclaration
- soit il y a un corps et il s'agit d'une définition

On n'utilise pratiquement jamais le mot-clé *extern* pour les fonctions.

Pour une variable c'est plus délicat car ambigu. Le problème ne se pose que pour les variables globales (i.e. non locales à une fonction ou un fichier).

Considérons un premier exemple :

```
int v = 0;
```

Cet exemple ne pose pas de problème : c'est une définition. Le fait d'initialiser une variable globale ne peut se faire que lors de sa définition.

Considérons un second exemple :

```
int v;
```

Est-ce une déclaration ? Ou est-ce une définition avec initialisation par défaut <sup>a</sup> à 0 ?

Le problème est que ce peut être les deux. En général le compilateur le devine seul, mais il me semble que parfois ce n'est pas le cas. En revanche si on compile avec le compilateur du C++ (`g++`) alors c'est systématiquement une définition.

Aussi prend-on l'habitude de faire précéder toutes les déclarations de variables globales par le mot-clé *extern*, ne serait-ce que pour que l'ensemble soit lisible. Les déclarations de variables globales sont quasi-systématiquement écrites dans les `.h` des modules.

Voici le code d'une déclaration :

```
extern int v;
```

L'exemple ci-dessus n'est pas ambigu : il ne peut s'agir que d'une déclaration.

Attention, *extern* ne signifie absolument pas qu'une variable est globale ; il signifie simplement que la variable est définie ailleurs. Ceci dit, comme nous l'avons vu, le mot *extern* n'a de sens que pour les variables globales ; mais il ne faut pas confondre la cause et l'effet.

Rappelons que les variables globales sont à éviter au maximum ; aussi est-on rarement confronté au problème.

<sup>a</sup>. Les variables globales et les variables *static* sont initialisées par défaut à 0, au contraire des variables locales qui n'ont pas d'initialisation par défaut. Ceci dit il est bien plus lisible d'initialiser explicitement une variable globale/*static* à 0.

## Fichiers d'entête .h

### 11.5.1 Définition

#### Point de cours

Un fichier .h (nommé également fichier d'entête) est la partie publique d'un module. Il contient généralement :

- des déclarations <sup>a</sup> de fonctions (très fréquent)
- des définitions de types
- des commentaires (quasi-systématiquement)
- des macros
- des déclarations de variables globales (très rare)

a. et non pas définition

### 11.5.2 Problématique des inclusions multiples

#### Point de cours

Un même fichier .h peut être inclus dans plusieurs fichiers .c, mais également dans d'autres fichiers .h. De fait il est possible d'inclure, dans un fichier .c, plusieurs fois le même .h sans s'en apercevoir.

Voyons cela avec un exemple simple et un .h système : *stdbool.h*. On suppose qu'on a un module de gestion de calendrier (*calendrier.h* et *calendrier.c*) avec notamment une fonction qui indique si une année est bissextile.

<pre>1 #include &lt;stdbool.h&gt; 2 3 bool estBissextile(int annee);</pre>	<pre>calendrier.c</pre> <pre>1 #include "calendrier.h" 2 3 bool estBissextile(int annee) 4 { ... }</pre>
--	---

Enfin nous considérons le programme principal, *main.c*, qui utilise le module calendrier, mais également des booléens par ailleurs, donc :

<pre>main.c</pre> <pre>1 #include &lt;stdbool.h&gt; 2 #include "calendrier.h" 3 4 ...</pre>
--

Et on voit que *stdbool.h* est inclus deux fois dans *main.c*, ce qui est potentiellement une catastrophe si *stdbool.h* définit des types (ce qui est bien entendu le cas).

Ce problème doit absolument avoir une solution, automatique si possible.

### 11.5.3 Solution non retenue

#### Point de cours

Nous continuons sur l'exemple du calendrier.

Puisque *stdbool.h* est déjà inclus dans *calendrier.h*, on n'inclut pas *stdbool.h* dans *main.c*.

Un premier problème est que si *main.c* n'a plus besoin du module des calendriers, il supprimera le *include* correspondant, et il faudra penser à remettre le *include* de *stdbool.h*.

Et de toute façon ce n'est pas satisfaisant car un programmeur n'a pas à connaître la liste des .h qu'il inclut indirectement. Par exemple, quand on inclut *stdio.h*, on inclut une quinzaine d'autres fichiers sans le savoir.

Cf. commandes pour avoir la liste des .h inclus :

tous les .h

```
gcc -M main.c
```

uniquement les .h non système

```
gcc -MM main.c
```

Donc cette solution n'est pas acceptable et n'est pas retenue.

#### 11.5.4 Solution retenue

##### Point de cours

Lorsqu'on inclut plusieurs fois un *.h* dans un *.c*, le *.h* visé se débrouille tout seul pour gérer et empêcher les inclusions multiples. C'est la solution retenue.

Le principe est d'englober chaque *.h* dans une macro conditionnelle qui gère notre problème. Voici le principe à appliquer pour tout *.h* :

```
toto.h  
1 #ifndef TOTO_H  
2 #define TOTO_H  
3  
4 // contenu du .h  
5  
6 #endif
```

La seule difficulté est de trouver un *define* différent pour chaque *.h*, c'est pourquoi on prend généralement le nom du fichier, par convention en majuscules.

Expliquons, sur un cas d'école avec le fichier *main.c*, pourquoi cela marche :

```
main.c  
1 #include "toto.h"  
2 #include "toto.h"  
3  
4 ...
```

Ce qui est équivalent à :

```
main.c  
1 #ifndef TOTO_H  
2 #define TOTO_H  
3  
4 // contenu du .h  
5  
6 #endif  
7 #ifndef TOTO_H  
8 #define TOTO_H  
9  
10 // contenu du .h  
11  
12 #endif  
13  
14 ...
```

- Ligne 1 : "si TOTO\_H n'est pas défini alors". C'est bien le cas puisqu'on n'a encore rien fait, donc on va exécuter les lignes 2 à 5.
- Ligne 2 : on définit *TOTO\_H*.
- Lignes 3 à 5 : toutes les lignes sont prises en compte.
- Ligne 7 : "si TOTO\_H n'est pas défini alors". Mais cette fois-ci *TOTO\_H* est défini (par la ligne 2) donc le test échoue et on passe directement à la ligne 12, et par conséquent on ne prend pas en compte une deuxième fois le fichier *toto.h*.

Note : tout ce mécanisme est exécuté par le préprocesseur et non par le compilateur. Autrement dit le compilateur ne voit pas les lignes 7 à 12.

### 11.5.5 Exercice

```
1 // pas de protections contre les doubles inclusions
2 // uniquement la structure
3
4 typedef struct
5 {
6     int x, y, z;
7 } vecteur;
```

**Listing 11.17** – code source de Vecteur.h sans les protections

```
1 // pas de protections contre les doubles inclusions
2 // il faut inclure Vecteur.h
3 // et définir la structure
4
5 #include "Vecteur.h"
6
7 typedef struct
8 {
9     vecteur arr[3];
10} repere;
```

**Listing 11.18** – code source de Repere.h sans les protections

```
1 #include "Vecteur.h"
2 #include "Repere.h"
3
4 int main()
5 {
6     return 0; // pas de EXIT_SUCCESS car on n'a pas "include <stdlib.h>"
7 }
```

**Listing 11.19** – code source de main.c sans les protections

```
crex@crex:~/.../5_5_EXERCICE/SANS_PROTECTION$ mygcc main main.c
In file included from Repere.h:5,
                 from main.c:2:
Vecteur.h:7:3: error: conflicting types for "vecteur"
  7 | } vecteur;
    | ^~~~~~
In file included from main.c:1:
Vecteur.h:7:3: note: previous declaration of "vecteur" was here
  7 | } vecteur;
    | ^~~~~~
```

**Listing 11.20** – Résultat de la compilation du main.c sans les protections

```

crex@crex:~/.../5_5_EXERCICE/SANS_PROTECTION$ gcc -E main.c
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "main.c"
# 1 "Vecteur.h" 1

typedef struct
{
    int x, y, z;
} vecteur;
# 2 "main.c" 2
# 1 "Repere.h" 1

# 1 "Vecteur.h" 1

typedef struct
{
    int x, y, z;
} vecteur;
# 6 "Repere.h" 2

typedef struct
{
    vecteur arr[3];
} repere;
# 3 "main.c" 2

int main()
{
    return 0;
}

```

**Listing 11.21** – Résultat du préprocesseur du main.c sans les protections

```

1 // pas de protections contre les doubles inclusions
2 // uniquement la structure
3
4 #ifndef VECTEUR_H
5 #define VECTEUR_H
6
7 typedef struct
8 {
9     int x, y, z;
10 } vecteur;
11
12 #endif

```

**Listing 11.22** – code source de Vecteur.h avec les protections

```

1 // pas de protections contre les doubles inclusions
2 // il faut inclure Vecteur.h
3 // et definir la structure
4 #ifndef REPERE_H
5 #define REPERE_H
6
7 #include "Vecteur.h"
8
9 typedef struct
10 {
11     vecteur arr[3];
12 } repere;
13
14 #endif

```

**Listing 11.23** – code source de Repere.h avec les protections

```

1 #include "Vecteur.h"
2 #include "Repere.h"
3
4 int main()
5 {
6     return 0; // pas de EXIT_SUCCESS car on n'a pas "include <stdlib.h>"
7 }

```

**Listing 11.24** – code source de main.c avec les protections

```

crex@crex:~/.../5_5_EXERCICE/AVEC_PROTECTION$ make
compiling main.c
creating main

crex@crex:~/.../5_5_EXERCICE/AVEC_PROTECTION$ ls
README  main  main.c  main.d  main.o  Makefile  Repere.h  Vecteur.h

```

**Listing 11.25** – Résultat de la compilation du main.c avec les protections

```
crex@crex:~/.../5_5_EXERCICE/AVEC_PROTECTION$ gcc -E main.c
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "main.c"
# 1 "Vecteur.h" 1

typedef struct
{
    int x, y, z;
} vecteur;
# 2 "main.c" 2
# 1 "Repere.h" 1
# 9 "Repere.h"
typedef struct
{
    vecteur arr[3];
} repere;
# 3 "main.c" 2

int main()
{
    return 0;
}
```

**Listing 11.26** – Résultat du préprocesseur du main.c avec les protections

### 11.5.6 Inclusions croisées

#### Point de cours

Voici un problème peu fréquent et pas toujours simple à résoudre : les inclusions croisées. On a une inclusion croisée lorsqu'un fichier *toto.h* inclut un fichier *titi.h* et réciproquement.

```
toto.h
1 #ifndef TOTO_H
2 #define TOTO_H
3
4 #include "titi.h" // car on utilise un type défini dans titi.h : myInt (ligne 7)
5
6 typedef float myFloat; // création d'un alias pour float
7 void g(myInt i);
8
9 #endif
```

```
titi.h
1 #ifndef TITI_H
2 #define TITI_H
3
4 #include "toto.h" // car on utilise un type défini dans toto.h : myFloat (ligne 7)
5
6 typedef int myInt; // création d'un alias pour int
7 void f(myFloat f);
8
9 #endif
```

```
main.c
1 #include "toto.h"
2 ...
3
```

### 11.5.7 Exercice

```
crex@crex:~/.../5_6_INCLUSIONS_CROISEES$ gcc -Wall -Wextra -pedantic -std=c99 -c main.c
In file included from toto.h:5,
from main.c:1:
titi.h:8:8: error: unknown type name "myFloat"; did you mean "float"?
  8 | void f(myFloat f);
     |          ^~~~~~
```

Réponse :

Juste pour illustrer le problème lorsque deux .h ont mutuellement besoin l'un de l'autre.  
Le code est réduit au strict minimum pour se focaliser sur le problème.

Ce n'est pas un problème fréquent, et quand cela arrive ce n'est pas toujours simple de s'en sortir.

Voici ce que donne main.c si on remplace les #include par le contenu des .h :

---

```
    == inclusion de toto.h dans main.c
01 #ifndef TOTO_H
02 #define TOTO_H
03
04     == inclusion de titi.h dans toto.h
05 #ifndef TITI_H
06 #define TITI_H
07
08     == inclusion de toto.h dans titi.h (le ifndef échoue)
09 #ifndef TOTO_H
10 #define TOTO_H
11     == code ignoré car TOTO_H est déjà défini (cf. ligne 2)
12 ...
13     #endif
14
15     #endif
16
17     typedef int myInt;      // création d'un alias pour int
18     void f(myFloat f);
19
20 #endif
```

---

Et on voit que la ligne 13 est incorrecte car myFloat n'est pas encore connu

Mettre les #include avant les #ifndef ne résout rien : soit c'est le même problème, soit il y a des inclusions infinies.

# Initialisation vs. affectation

## 11.6.1 Explications

### Point de cours

Initialiser une variable signifie : “mettre une valeur lors de la définition”.

Affecter une variable signifie : “remplacer la valeur courante par une autre”.

Il y a souvent des abus de langage où l'on emploie un mot pour l'autre.

Note : cette différence entre ”initialisation” et ”affectation” prendra une grande importance en programmation objet (peut-être plus particulièrement en C++) avec les constructeurs et les opérateurs d'affectation.

Voici un premier code :

```
toto.h  
1 int a = 3;  
2 a = 7;
```

- Ligne 1 : *a* est définie et initialisée à 3.
- Ligne 2 : *a* est affectée (la valeur 3 est remplacée par 7).

Voici un second code :

```
toto.h  
1 int a;  
2 a = 7;
```

- Ligne 1 : *a* est définie et soit initialisée à 0 si c'est une variable globale, soit ”initialisée” à une valeur ”aléatoire” pour une variable locale.
- Ligne 2 : *a* est affectée (sa valeur est remplacée par 7).
- Si *a* est une variable locale, par abus on dit qu'elle est initialisée à 7 lors de la ligne 2. C'est d'ailleurs conceptuellement vrai (i.e. c'est bien le sens du programme) mais c'est techniquement faux.

Et pour les constantes ? Par définition une constante doit être initialisée et ne peut pas être affectée.

### 11.6.2 Exercice

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // définir ici une variable non static
5 int i;
6 // définir ici une variable static
7 static int j;
8
9 void f()
10 {
11     // définir ici une variable static
12     static int k;
13     // définir ici une variable non static
14     int l;
15
16     // afficher les deux variables
17     printf("k: %i , l: %i\n", k, l);
18 }
19
20 int main()
21 {
22     // définir ici une variable non static
23     int m;
24
25     // afficher les 3 variables visibles du main
26     printf("i: %i , j: %i , m: %i\n", i, j, m);
27     f();
28
29     return EXIT_SUCCESS;
30 }
```

Listing 11.27 – code source de l'exercice 6.2

```
(base) florian@CReXJr:~/.../6_2_EXERCICE$ make
compiling main.c
main.c: In function "f":
main.c:17:5: warning: "l" is used uninitialized in this function [-Wuninitialized]
    17 |         printf("k: %i , l: %i\n", k, l);
                  ^_____
main.c: In function "main":
main.c:26:5: warning: "m" is used uninitialized in this function [-Wuninitialized]
    26 |         printf("i: %i , j: %i , m: %i\n", i, j, m);
                  ^_____
creating main
```

Listing 11.28 – Résultat de la compilation du code source de l'exercice 6.2

```
(base) florian@CRexJr:~/.../6_2_EXERCICE$ ./main
i: 0, j: 0, m: 0
k: 0, l: 21846
```

**Listing 11.29** – Résultat de l'exécution de la compilation du code source de l'exercice 6.2

**Réponse :** La variable locale "l" non static à f() et la variable locale "m" non static à la fonction main() font l'objet d'un warning (le warning nous avertit qu'elles ne sont pas initialisées). Comme indiqué dans l'exercice 4 de ce TP en note de bas de page, certaines variables ont une initialisation par défaut à 0. Il semble que ce soit malgré tout le cas pour la variable non static locale à la fonction main(), mais pas pour "l". Toutes les autres variables ont une initialisation par défaut à 0.

### 11.6.3 Exercice

**Question :** Définissez, sans les initialiser, 4 constantes (mot-clé const) entières :

- une constante globale
- une constante static hors des fonctions
- une constante static dans une fonction autre que le main
- une constante non static dans une fonction autre que le main
- une constante non static dans la fonction main

Notez quelles constantes sont concernées par un warning, et expliquez. Au fait, pourquoi définir une constante static dans une fonction n'a pas d'intérêt ? Essayez d'affecter chacune des 5 constantes et regardez les messages du compilateur.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // définir ici une constante non static (sans l'initialiser)
5 const int i;
6 // définir ici une constante static (sans l'initialiser)
7 static const int j;
8
9 void f()
10 {
11     // définir ici une constante static (sans l'initialiser)
12     static const int k;
13
14     // définir ici une constante non static (sans l'initialiser)
15     const int l;
16
17     // essayez de changer ces deux constantes
18     //k = 0;
19     //l = 1;
20
21     // afficher les deux constantes
22     printf("k: %i, l: %i\n", k, l);
23 }
24
25 int main()
26 {
27     // définir ici une constante non static (sans l'initialiser)
28     const int m;
29
30     // afficher les 3 constantes visibles du main
31     // essayez de changer ces trois constantes
32     printf("i: %i, j: %i, m: %i\n", i, j, m);
33     //i = 2;
34     //j = 3;
35     //m = 4;
36     f();
37
38     return EXIT_SUCCESS;
39 }
```

**Listing 11.30** – code source de l'exercice 6.3

```

crex@crex:~/.../6_3_EXERCICE$ make
compiling main.c
main.c: In function "f":
main.c:22:5: warning: "l" is used uninitialized in this function [-Wuninitialized]
22 |     printf("k: %i , l: %i\n" , k, l);
|           ^
main.c: In function "main":
main.c:32:5: warning: "m" is used uninitialized in this function [-Wuninitialized]
32 |     printf(" i: %i , j: %i , m: %i\n" , i, j, m);
|           ^
creating main

crex@crex:~/.../6_3_EXERCICE$ ./main
i: 0, j: 0, m: 0
k: 0, l: 22003

```

**Listing 11.31** – Résultat de la compilation sans les affectations des constantes

**Réponse :** On remarque alors que les résultats sont les mêmes que ceux de l'exercice 6.2. Le mot clef constante n'influe pas sur les résultats.

**Remarque :** Une constante static dans une fonction n'a pas d'intérêt car la valeur d'une constante ne change pas. Or l'utilité d'une variable static à une fonction est qu'elle soit persistante et qu'on puisse éventuellement la modifier.

```

crex@crex:~/.../6_3_EXERCICE$ make
compiling main.c
main.c: In function "f":
main.c:18:7: error: assignment of read-only variable "k"
  18 |     k = 0;
      |
main.c:19:7: error: assignment of read-only variable "l"
  19 |     l = 1;
      |
main.c: In function "main":
main.c:33:7: error: assignment of read-only variable "i"
  33 |     i = 2;
      |
main.c:34:7: error: assignment of read-only variable "j"
  34 |     j = 3;
      |
main.c:35:7: error: assignment of read-only variable "m"
  35 |     m = 4;
      |
make: *** [Makefile:117: main.o] Error 1

```

**Listing 11.32** – Résultat de la compilation avec affectation des constantes

**Réponse :** On ne peut pas changer les valeurs de constantes, elles sont donc pour la machine des variables "read-only", ie. en lecture seule.

## Exercice complet

### 11.7.1 Module “mathématiques financières”

#### Question :

On a besoin de faire des opérations sur les comptes bancaires, en l'occurrence des additions et des soustractions (!).

Nous n'allons pas utiliser les opérateurs + et -, mais des fonctions dédiées.

Il faut créer le module operation avec les fichiers operation.h et operation.c.

Deux fonctions principales sont à définir :

- int plus(int a, int b) qui renvoie la somme de a et b ;
- int moins(int a, int b) qui renvoie la différence de a et b.

En outre, il s'agit de définir la fonction :

- int getNbOperations() qui renvoie le nombre d'appels à plus et moins, c'est-à-dire qui compte le nombre d'appels effectués aux opérations plus et moins.

De manière gratuite, ce module contiendra une variable globale dummy entière qui ne sert à rien. Elle sera initialisée à 14 à sa création. Les autres modules peuvent la manipuler à leur guise.

Créez les fichiers operation.h et operation.c.

#### Réponse :

```
1 #ifndef OPERATION_H
2 #define OPERATION_H
3
4 //Returns the sum of a and b
5 int plus(int a, int b);
6
7 //Returns the subtraction of a and b
8 int moins(int a, int b);
9
10 //Counts the number of times plus() and moins() have been called
11 //Returns that counter
12 int getNbOperations();
13
14 //Functions to manipulate global variable int dummy in operation.c
15 extern int dummy;
16
17 #endif
```

**Listing 11.33** – code source de operation.h

```
1 #include "operation.h"
2
3 int plus(int a, int b)
4 {
5     getNbOperations();
6     return a + b;
7 }
8
9 int moins(int a, int b)
10 {
11     getNbOperations();
12     return a - b;
13 }
14
15 int getNbOperations()
16 {
17     static int n = 0;
18     n++;
19     return n;
20 }
21
22 int dummy = 14;
```

**Listing 11.34** – code source de operation.c

### 11.7.2 Module “gestion de comptes”

**Question :** Il faut créer le module comptes avec les fichiers comptes.h et comptes.c. Ce module gère les comptes bancaires d'une agence. Il se trouve que le nombre de comptes est fixe (et immuable), par exemple 10, mais ce nombre doit être modifiable facilement. Pour l'énoncé, N sera le nombre de comptes. Les comptes sont numérotés de 0 à N-1. Les comptes ne sont pas initialisés à 0 euro, mais à un montant fixé dans le programme (par exemple 15 euros). Les fonctions accessibles à l'utilisateur du module sont :

- int getNbComptes()
- int getSoldeCompte(int numCompte)
- void creditCompte(int numCompte, int montant)
- void debitCompte(int numCompte, int montant)

Ce module ne donne pas accès à d'autres fonctions et les prototypes sont imposés. Les fonctions creditCompte et debitCompte n'ont pas le droit d'utiliser + et -, mais doivent utiliser les fonctions plus et moins du module operation.

La fonction creditCompte est particulière. En règle générale elle ajoute au compte désigné le montant fourni. Mais tous les 7 appels (encore fois c'est une constante qui doit être facilement paramétrable) elle ajoute 1 en plus du montant ; c'est une sorte de publicité que fait la banque. Notez que ce compteur est commun à tous les comptes. On pourrait imaginer avoir un compteur par compte bancaire (ce qui serait plus logique), mais ce n'est pas demandé.

En revanche les N comptes bancaires se sont pas accessibles directement hors du fichier comptes.c ; il faut passer par les quatre fonctions décrites ci-dessus.

Créez les fichiers comptes.h et comptes.c.

**Réponse :**

```
1 #ifndef COMPTES_H
2 #define COMPTES_H
3
4 int getNbComptes();
5 int getSoldeCompte(int numCompte);
6 void creditCompte(int numCompte, int montant);
7 void debitCompte(int numCompte, int montant);
8
9 #endif
```

Listing 11.35 – code source de comptes.h

```
1 // todo : des include
2 #include "operation.h"
3 #include "comptes.h"
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 // todo : définitions des constantes, types, variables, ...
8 #define N 10
9
10 //static const int N = 10;
11 static const int CREDIT_TIME = 7;
12 static const int CREDIT_BONUS = 1;
13
14 static int comptes[N];
```

```

15 /*
16 * fonctions internes
17 */
18
19 // gestion violente
20 // l'utilisation d'un assert serait plus élégante
21 // todo : la fonction est locale au fichier, y a-t-il quelque chose à faire
22 ?
23 static void verifNumCompte(int numCompte)
24 {
25     if ((numCompte < 0) || (numCompte >= getNbComptes()))
26     {
27         fprintf(stderr, "numéro de compte incorrect.\n");
28         exit(EXIT_FAILURE);
29     }
30 }
31
32 // todo : le reste du code
33
34 int getNbComptes()
35 {
36     return N;
37 }
38
39 int getSoldeCompte(int numCompte)
40 {
41     verifNumCompte(numCompte);
42     return comptes[numCompte];
43 }
44
45 void creditCompte(int numCompte, int montant)
46 {
47     verifNumCompte(numCompte);
48     static int count = 0;
49
50     if (count == CREDIT_TIME)
51         comptes[numCompte] = plus(comptes[numCompte], montant +
CREDIT_BONUS);
52
53     else
54         comptes[numCompte] = plus(comptes[numCompte], montant);
55
56     count++;
57 }
58
59 void debitCompte(int numCompte, int montant)
60 {
61     verifNumCompte(numCompte);
62     comptes[numCompte] = moins(comptes[numCompte], montant);
63 }
```

**Listing 11.36** – code source de comptes.c

### 11.7.3 Module du programme principal : main

**Remarque :** Il n'y a aucune raison de placer la fonction main dans le module operation ou comptes. En effet ces modules peuvent être utilisés par d'autres programmeurs qui auront leur propre main.

Il faut donc un module spécifique cette fonction. On note qu'il n'y a pas de .h correspondant. Le fichier main.c est fourni à l'exception des include des deux modules à ajouter.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // INCLURE ICI LES DEUX .h
5 #include "operation.h"
6 #include "comptes.h"
7
8 /*
9 * Juste pour vérifier le module "operation"
10 */
11 void exo1()
12 {
13     int r;
14
15     printf("dummy = %d\n", dummy);
16     dummy += 19;
17     printf("dummy = %d\n", dummy);
18
19     r = plus(3, 4);
20     printf("la somme de 3 et 4 est %d\n", r);
21
22     r = moins(3, 4);
23     printf("la soustraction de 3 et 4 est %d\n", r);
24 }
25
26 /*
27 * manipulation des comptes bancaires
28 */
29 void exo2()
30 {
31     printf("Il y a %d compte(s) bancaire(s)\n", getNbComptes());
32
33     // quelques crédits (rappel : les comptes ne sont pas initialisés à 0)
34     creditCompte(0, 1);
35     creditCompte(1, 10);
36     creditCompte(2, 100);
37     creditCompte(3, 1000);
38     creditCompte(4, 10000);
39     creditCompte(0, 1);
40     creditCompte(1, 10);           // on profite du bonus (si tous les 7 crédits)
41     creditCompte(2, 100);
42     creditCompte(3, 1000);
43     creditCompte(4, 10000);
44
45     // affichage complet
46     printf("Etats des comptes : \n");
```

```

47 for (int i = 0; i < getNbComptes(); i++)
48     printf("Compte %2d : %5d euro(s)\n", i, getSoldeCompte(i));
49 printf("\n");
50
51 // un débit tout de même
52 debitCompte(2, 7);
53 printf("Compte %2d : %5d euro(s)\n", 2, getSoldeCompte(2));
54
55 // s'il y a moins de 1000 comptes, ça plante
56 // creditCompte(999, 1);
57
58 printf("Il y a eu %d opérations\n", getNbOperations());
59 }
60
61 int main()
62 {
63     exo1();
64     exo2();
65
66     return EXIT_SUCCESS;
67 }
```

**Listing 11.37** – code source de main.c

#### 11.7.4 Compilation

Solution 1 : (à éviter en général)

```
$ gcc -Wall -Wextra -pedantic -std=c99 -g *.c -o main
```

Solution 2 : (mieux)

```
$ gcc -Wall -Wextra -pedantic -std=c99 -g operation.c comptes.c main.c -o main
```

Solution 3 : (idéale mais ingérable sans Makefile)

```
$ gcc -Wall -Wextra -pedantic -std=c99 -g -c operation.c  
$ gcc -Wall -Wextra -pedantic -std=c99 -g -c comptes.c  
$ gcc -Wall -Wextra -pedantic -std=c99 -g -c main.c  
$ gcc -Wall -Wextra -pedantic -std=c99 -g operation.o comptes.o main.o -o main
```

Solution 4 : avec un Makfile (qui est fourni !)

```
$ make
```

La solution 4 semble la plus adéquate.

```
(base) florian@CReXJr:~/.../7_EXERCICE$ make  
compiling main.c  
compiling operation.c  
compiling comptes.c  
creating main
```

**Listing 11.38** – Compilation du projet de programme bancaire

```
(base) florian@CReXJr:~/.../7_EXERCICE$ ./main  
dummy = 14  
dummy = 33  
la somme de 3 et 4 est 7  
la soustraction de 3 et 4 est -1  
Il y a 10 compte(s) bancaire(s)  
Etats des comptes :  
Compte 0 : 2 euro(s)  
Compte 1 : 20 euro(s)  
Compte 2 : 201 euro(s)  
Compte 3 : 2000 euro(s)  
Compte 4 : 20000 euro(s)  
Compte 5 : 0 euro(s)  
Compte 6 : 0 euro(s)  
Compte 7 : 0 euro(s)  
Compte 8 : 0 euro(s)  
Compte 9 : 0 euro(s)  
  
Compte 2 : 194 euro(s)  
Il y a eu 14 opérations
```

**Listing 11.39** – Résultat de l'exécution du projet bancaire

# Chapitre 12

## Fonctions de gestions de fichiers en C

### Résumé et cadre de travail

Le but de cette description est d'avoir une vue d'ensemble de la manipulation des fichiers en C, et notamment :

- la notion d'entrées/sorties bufferisées
- la différence entre mode binaire et mode texte

Une suite d'exercices permet d'appliquer tous ces concepts.

### Mode bufferisé ou non

#### Point de cours

Il y a deux familles de fonctions pour manipuler les fichiers :

- open close lseek write read dprintf ...
- fopen fclose fseek fwrite fread fprintf fscanf fflush ...

#### *Famille open :*

Ce sont des appels système a priori moins portables (n'existent pas sous Windows je pense). Les entrées/sorties sont bas niveau, i.e. non bufferisées.

#### *Famille fopen :*

Ce sont des fonctions de la bibliothèque standard du C et donc plus portables. Les entrées/sorties sont plus haut niveau, i.e. bufferisées.

Des écritures bufferisées ne sont pas écrites directement sur disque mais stockées en mémoire. Ce sont les fonctions qui décident quand écrire sur disque (buffer plein, fermeture du fichier, ...). C'est bien plus efficace (moins d'accès disque et avec des volumes de données plus grands), mais on maîtrise moins ce que l'on fait.

Des écritures non bufferisées sont immédiatement écrites sur disque avec une inversion des avantages et inconvénients par rapport au mode bufferisé.

# Mode binaire ou mode texte

## Point de cours

Lorsqu'on écrit en mode "texte", le résultat est compréhensible par un humain, i.e. le fichier peut être ouvert avec un éditeur de texte.

En mode binaire, on écrit les données selon le codage de la machine et il faut un logiciel dédié pour lire le fichier (style *od*).

Avantages du mode texte :

- lisible par l'humain
- portable d'une architecture matérielle à une autre

Inconvénients du mode texte :

- nécessite des conversions de codage
- la place occupée par les données peut dépendre des valeurs de ces données
- relecture des données par programme plus difficile (du fait de la taille variable)

Voyons par l'exemple avec l'écriture d'un int.

En mode texte (avec `fprintf` ou `dprintf`) :

- il est converti en suite de caractères qui sont mis les uns à la suite des autres dans le fichier (123 sera transformé en '1' '2' et '3')
- le nombre de caractères écrits va dépendre de la valeur (3 caractères pour 123, 7 caractères pour -123456, ...)

En mode binaire (avec `fwrite` ou `write`) :

- c'est directement l'image mémoire du nombre qui est écrite
- la taille est fixe quelle que soit la valeur (4 pour un int, 8 pour un long)

Et pour la lecture

En mode texte :

- il faut faire la conversion inverse
- il faut savoir combien de caractères lire
- le fichier peut être lu sur d'autre architectures

En mode binaire

- recopie des valeurs du fichier directement en mémoire
- on sait combien d'octets il faut lire

Ecriture de deux nombres

En mode texte

- il faut les séparer par un séparateur (sic) : espace ou autre en effet si on écrit 12 et 34 sans précaution, dans le fichier il y aura "1234" et impossible lors de la lecture de savoir quoi lire (1 et 234 ? 12 et 34 ? 123 et 4 ?)

En mode binaire

- il n'y a pas de soucis, pour les int les 4 premiers octets contiennent le premier nombre, les 4 suivants le second

Lecture d'un tableau de nombres, on veut récupérer le 13me

En mode texte

- il faut lire les nombres un par un en sautant les espaces jusqu'à arriver au 13me

En mode binaire

- on se déplace (pour les int) à la position 48 ( $4 * (13 - 1)$ )
- et on lit 4 octets

Portabilités inter-architecture

Prenons l'exemple des int : selon les architectures l'ordre des 4 octets est en big endian ou little endian.

Big endian : les octets sont ordonnés du poids fort vers le poids faible

Little endian : c'est le contraire

Exemple : 3270897 (en hexa : 0031E8F1)

- les valeurs des quatre octets sont : 0 49 232 241
- en big endian, les octets sont dans cet ordre en mémoire
- en little endian, l'ordre est 241 232 49 0

Si on écrit le fichier sur une architecture et qu'on le lit sur une qui est inversée, c'est la catastrophe. Si on écrit 3270897 en little endian, et qu'on le lit en big endian, on obtiendra -236441344.

(cf. fonctions `htonl`, `hton`, `ntohl`, `ntoh` si vous êtes intéressés)

## Liste de quelques fonctions

### Point de cours

#### *Haut-niveau (bufferisé) :*

- fopen : ouverture
- fclose : fermeture
- fwrite : écriture en binaire
- fread : lecture en binaire
- fprintf : écriture en mode texte
- fscanf : lecture en mode texte
- fseek : se déplacer dans le fichier
- fflush : forcer l'écriture du buffer sur disque

#### *Bas-niveau (non bufferisé) :*

- open : ouverture
- close : fermeture
- write : écriture en binaire
- read : lecture en binaire
- dprintf : écriture en mode texte
- lseek : se déplacer dans le fichier

## Exercices

### 12.5.1 Exercice A

#### Question :

Ouverture et fermeture de fichiers : le but est d'effectuer des manipulations de base :

- Avec open, dprintf et close, créer le fichier exoA 1 test et écrire l'entier 12 dedans.
- Avec open, write et close, créer le fichier exoA 2 test et écrire l'entier 12 dedans.
- Avec fopen, fprintf et fclose, créer le fichier exoA 3 test et écrire l'entier 12 dedans.
- Avec fopen, fwrite et fclose, créer le fichier exoA 4 test et écrire l'entier 12 dedans.

#### Réponse :

```
1 #define _POSIX_C_SOURCE 200809L
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/types.h> //Requis pour open , cf . man open
6 #include <sys/stat.h> //Requis pour open
7 #include <fcntl.h> //Requis pour open
8 #include <unistd.h> //Requis pour close
9
10 int main()
11 {
12     // avec open , dprintf et close , créer le fichier "exoA_1_test" et é
13     //crire l'entier 12 dedans
14     //int fd = open("./exoA_1_test", O_CREAT | O_WRONLY,
15     //              //S_IRWXU | S_IRWXG | S_IROWXO);
16     int fd = open("./exoA_1_test", O_CREAT | O_WRONLY, 00644);
17     dprintf(fd, "%i", 12);
18     close(fd);
19
20     // avec open , write et close , créer le fichier "exoA_2_test" et écrire
21     // l'entier 12 dedans
22     int fd2 = open("./exoA_2_test", O_CREAT | O_WRONLY, 00644);
23     char buf[]="12";
24     write(fd2, buf, sizeof(buf));
25     close(fd2);
26
27     // avec fopen , fprintf et fclose , créer le fichier "exoA_3_test" et é
28     //crire l'entier 12 dedans
29     FILE *fp = fopen("./exoA_3_test", "w+");
30     fprintf(fp, "%i", 12);
31     fclose(fp);
32
33     // avec fopen , fwrite et fclose , créer le fichier "exoA_4_test" et é
34     //crire l'entier 12 dedans
35     FILE *fp2 = fopen("./exoA_4_test", "w+");
36     int v = 12;
37     fwrite(&v, sizeof(int), 1, fp2);
38     fclose(fp2);
39
40     return EXIT_SUCCESS;
41 }
```

**Question :**

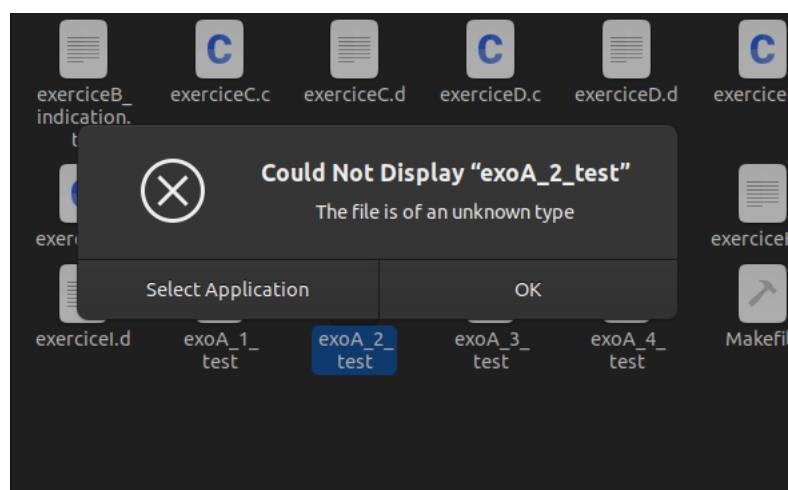
Examinez les 4 fichiers générés :  
— avec un éditeur de texte

**Réponse :**

- 1) On constate que les fichiers écrits par des fonctions binaires n'apparaissent pas comme des fichiers textes du point de vue du système d'exploitation...



De plus, ils ne peuvent pas être ouverts avec un éditeur de texte classique à moins de prendre le risque de corrompre les données du fichier (ici on a essayé "gedit").

**Question :**

Examinez les 4 fichiers générés :  
— avec la commande : od -Ad -w10 -t d1z <nom fichier>  
— avec la commande : od -Ad -w10 -t x1z <nom fichier>  
Comprenez et expliquer la signification de chaque octet.

**Résultats :**

- 1) Avec la commande : od -Ad -w10 -t d1z <nom fichier> on obtient le résultat suivant :

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ od -Ad -w10 -t d1z exoA_1_test
0000000 49 50 >12<
0000002
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ od -Ad -w10 -t d1z exoA_2_test
0000000 49 50 0 >12.<
0000003
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ od -Ad -w10 -t d1z exoA_3_test
0000000 49 50 >12<
0000002
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ od -Ad -w10 -t d1z exoA_4_test
0000000 12 0 0 0 >....<
0000004
```

2) Avec la commande : od -Ad -w10 -t x1z <nom fichier>

```
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ od -
      Ad -w10 -t x1z exoA_1_test
00000000 31 32                                >12<
00000002
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ od -
      Ad -w10 -t x1z exoA_2_test
00000000 31 32 00                               >12.<
00000003
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ od -
      Ad -w10 -t x1z exoA_3_test
00000000 31 32                                >12<
00000002
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ od -
      Ad -w10 -t x1z exoA_4_test
00000000 0c 00 00 00                           >....<
00000004
```

#### Explications préliminaires :

L'option `-w<number>` permet d'indiquer combien d'octets on veut afficher à chaque ligne. Le nombre d'octets affichés par ligne se reflète dans la première colonne qui indique le nombre d'octets qui ont été affichés depuis le début du fichier jusqu'à cette ligne.

L'option `-A[d/o/x/n]` permet d'indiquer le format avec lequel on veut que cette première colonne s'affiche. Ce format est respectivement en base 10 (d pour décimal), en octal, en hexadécimal ou bien on veut que la première colonne ne soit pas affichée (n pour none).

L'option `-t <format>` permet d'indiquer dans quel format on veut qu'un octet ou un ensemble d'octets sur chaque ligne s'affiche. Par exemple : `-w10 -t d1z` signifie que chaque ligne contient 10 octets et 1 octet (le 1 de d1z...) s'affichera en base 10. Le "z" permet de dire qu'on souhaite un affichage de la ligne considérée en caractères imprimables quelconques (en ascii par exemple). Si on avait écrit : `-w10 -t x2z` alors on aurait signifié qu'on souhaite afficher tous les deux octets leur valeur concaténée en hexadécimale.

La commande `od` elle-même signifie qu'on souhaite un affichage du fichier considéré en octets avec par défaut un affichage en octal pour tous les deux octets concaténés (cf. Dans le manuel de Bash : "od -A o -t oS -w16 The default output format used by od", le S majuscule signifiant sizeof(short) sous-entendu "short int"... Un short int est sur deux octets)

Réponse :

```
1 #define _POSIX_C_SOURCE 200809L
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/types.h> //Requis pour open , cf . man open
6 #include <sys/stat.h> //Requis pour open
7 #include <fcntl.h> //Requis pour open
8 #include <unistd.h> //Requis pour close
9
10 int main()
11 {
12     // avec open , dprintf et close , créer le fichier "exoA_1_test" et écrire l'entier 12 dedans
13
14     //int fd = open("./exoA_1_test", O_CREAT | O_WRONLY,
15     //              S_IRWXU | S_IRWXG | S_IROWXO);
16     int fd = open("./exoA_1_test", O_CREAT | O_WRONLY, 00644);
17     dprintf(fd, "%i", 12);
18     close(fd);
19
20     // avec open , write et close , créer le fichier "exoA_2_test" et écrire l'entier 12 dedans
21
22     int fd2 = open("./exoA_2_test", O_CREAT | O_WRONLY, 00644);
23     char buf[]="12";
24     write(fd2, buf, sizeof(buf));
25     close(fd2);
26
27     // avec fopen , fprintf et fclose , créer le fichier "exoA_3_test" et écrire l'entier 12 dedans
28     FILE *fp = fopen("./exoA_3_test", "w+");
29     fprintf(fp, "%i", 12);
30     fclose(fp);
31
32     // avec fopen , fwrite et fclose , créer le fichier "exoA_4_test" et écrire l'entier 12 dedans
33     FILE *fp2 = fopen("./exoA_4_test", "w+");
34     int v = 12;
35     fwrite(&v, sizeof(int), 1, fp2);
36     fclose(fp2);
37
38     return EXIT_SUCCESS;
39 }
```

1) À partir de ces explications préliminaires et du code ci-dessus on comprend mieux les résultats précédents. La commande en 1) demande à ce que la première colonne indiquant combien d'octets ont été affichés depuis le début du fichier soit au format décimal. On souhaite également que chaque ligne comporte 10 octets et que chaque octet soit affiché au format décimal avec un affichage de chaque ligne en caractères imprimables.

Le fichier exoA\_1\_test a été écrit avec la fonction dprintf une fonction d'écriture en mode texte non bufferisée. On a donc les valeurs ascii 49 (correspondant au 1 ascii) et 50 (correspondant au 2 ascii) pour chaque octet. Des colonnes vides séparent ces deux octets de l'affichage en caractères imprimables >12<. Ce sont des octets inexistant dans le fichier mais comme on demande 10 octets par ligne la commande

od laisse des colonnes vides. La commande od termine son affichage avec une ligne vide après les deux octets du fichier...

Pour le fichier exoA\_2\_ test on utilise la fonction write, fonction non bufferisée d'écriture en binaire. On utilise pour cela une chaîne de caractères contenu dans un tableau "char buf[]". Le null byte "\0" est donc écrit dans le fichier et apparaît comme étant un octet supplémentaire valant "00" et un point dans l'affichage en caractères imprimables (le point termine une phrase...)

Pour le fichier exoA\_3\_ test on utilise la fonction fprintf, fonction bufferisée d'écriture en mode texte. On se rend compte que, bufferisé ou non, le résultat dans un fichier est le même pour des écritures en mode texte ou binaire.

La différence d'affichage pour le fichier exoA\_4\_ test tient surtout au fait que cette-fois si on n'utilise plus un tableau de caractères mais directement un entier dont la valeur apparaît bien comme 12 dans les octets du fichier mais qui ne produit aucun affichage en caractères imprimables car 12 en ascii signifie [FORM FEED] comme expliqué dans le document ci-dessous...

## ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	I	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	-	127	7F	[DEL]

**Question :**

Examinez les 4 fichiers générés :

- avec la commande : hexdump -C <nom fichier>
- avec la commande : hd <nom fichier>

Comprenez et expliquer la signification de chaque octet.

**Résultats :** 1) Avec la commande : hexdump -C <nom fichier>

```
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ hexdump -C exoA_1_test  
00000000 31 32 | 12 |  
00000002  
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ hexdump -C exoA_2_test  
00000000 31 32 00 | 12 . |  
00000003  
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ hexdump -C exoA_3_test  
00000000 31 32 | 12 |  
00000002  
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ hexdump -C exoA_4_test  
00000000 0c 00 00 00 | . . . |  
00000004
```

2) Avec la commande : hd <nom fichier>

```
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ hd  
exoA_1_test  
00000000 31 32 | 12 |  
00000002  
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ hd  
exoA_2_test  
00000000 31 32 00 | 12 . |  
00000003  
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ hd  
exoA_3_test  
00000000 31 32 | 12 |  
00000002  
(base) florian@CRegJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP4/CODE_FOURNI$ hd  
exoA_4_test  
00000000 0c 00 00 00 | . . . |  
00000004
```

**Réponse :**

La fonction hexdump réalise (bien qu'avec une syntaxe différente) à peu près les mêmes fonctions que "od" (qui peut se lire comment octaldump...) mais en prenant le format hexadecimal comme format par défaut pour les affichages de la première colonne et des octets sur chaque ligne. De même, l'option -C permet de préciser combien d'octets nous voulons pour chaque ligne et tous les combien d'octets on fait une concaténation (ici on concatène tous les 2 octets et on affiche 32 octets par ligne.) L'option -C précise aussi qu'on souhaite un affichage en caractères imprimables entre deux barres verticales...

En prenant en compte ces informations on remarque que les affichages révèlent les mêmes choses que ceux de la commande "od" avec un format différent...

On notera que "hd" est un raccourci pour "hexdump -C <nomFichier>"

### 12.5.2 Exercice B : curiosité binaire

**Question :** Créez le fichier exoB 1 test et écrivez dedans en binaire (avec fwrite) le nombre n1 = 1952540739. De la même façon, créez le fichier exoB 2 test avec le nombre n2 = 1130914164. Ouvrez les fichiers générés avec un éditeur de texte et expliquez.

Résultats :

The image shows two terminal windows side-by-side. The top window is titled "exoB\_1\_test" and contains the binary string "01000011 01101000 01100001 01110100". The bottom window is titled "exoB\_2\_test" and contains the binary string "01110100 01100001 01101000 01000011". Both windows are dark-themed with white text.

Réponse :

C            h            a            t  
01000011 01101000 01100001 01110100  
(1952540739)<sub>10</sub> = (01110100 01100001 01101000 01000011)<sub>2</sub>

---

(1130914164)<sub>10</sub> = (01000011 01101000 01100001 01110100)<sub>2</sub>  
01110100 01100001 01101000 01000011  
t            a            h            C

### 12.5.3 Exercice C : écriture/lecture d'une chaîne de caractères

Question :

Écrire une fonction ;

*void ecritChaine(const char \*nomFichier, const char \*s)*

qui sauvegarde une chaîne de caractères dans un fichier.

Écrire une fonction :

*void litChaine(const char \*nomFichier, char \*s)*

qui lit une chaîne de caractères dans un fichier (chaîne écrite par la fonction précédente). Il ne faut pas effectuer la lecture caractère par caractère, mais lire la chaîne d'un bloc, autrement dit en une seule instruction (il n'est pas interdit de stocker des informations supplémentaires dans le fichier lors de l'écriture). Note : on stocke le résultat dans s qui doit être suffisamment grande. Il est de la responsabilité de l'appelant de s'en assurer ; la fonction ne doit pas (et ne peut pas d'ailleurs) le faire.

Dans la fonction main, faites le code qui écrit une chaîne stockée dans une variable dans un fichier et qui relit cette même chaîne dans une autre variable, puis qui compare les valeurs de ces deux variables.

Réponse :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h> // Nécessaire car fonction strcmp...
4
5
6 void ecritChaine(const char *nomFichier, const char *s)
7 {
8     FILE *f = fopen(nomFichier, "w+");
9     int s_len = strlen(s);
10    fprintf(f, "%i %s", s_len, s);
11    fclose(f);
12 }
13
14 void litChaine(const char *nomFichier, char *s)
15 {
16     FILE *f = fopen(nomFichier, "r");
17     int s_len = 0;
18     fscanf(f, "%i ", &s_len);
19     fread(s, sizeof(char), s_len, f);
20     fclose(f);
21 }
22
23 int main()
24 {
25     // note : on ne gère pas ici la mémoire, ce n'est pas le but de l'exercice
26     char original[1000] = "Bonjour monde !";
27     char copie[1000];
28
29     // todo (cf. spoiler ci-dessous)
30     ecritChaine("./exoC_test", original);
31     litChaine("./exoC_test", copie);
32
33
34     if(strcmp(original, copie) == 0)
35         printf("Les chaînes sont identiques\n");
```

```
36
37     else
38         printf( "ERREUR => %s :::: %s\n" , original , copie );
39     return EXIT_SUCCESS;
40 }
```

#### 12.5.4 Exercice D : string input/output (bis repetita)

##### Question :

On reprend le même problème que l'exercice précédent, mais on écrit et lit un tableau de chaînes de caractères.

Écrire une fonction ;

```
void ecritTabChaines(const char *nomFichier, char * s[], int size)
```

qui sauvegarde un tableau de chaînes de caractères dans un fichier.

Écrire une fonction ;

```
void litTabChaines(const char *nomFichier, char * tab[], int *size)
```

qui lit un tableau de chaînes de caractères dans un fichier (tableau écrit par la fonction précédente).

Comment savoir le nombre de chaînes à lire ?

Comment faire pour minimiser le nombre d'instructions de lecture : chaque chaîne doit être lue en une seule instruction (il n'est pas interdit de stocker des informations supplémentaires dans le fichier lors de l'écriture) ? ?

Notes :

- le tableau tab doit être suffisamment grand : c'est de la responsabilité de l'appelant de le pré-allouer ;
- size contiendra le nombre de chaînes lues.

Dans la fonction main, faites le code qui écrit un tableau stocké dans une variable dans un fichier et qui relit ce même tableau dans une autre variable, puis qui compare les valeurs de ces deux variables. Le code pour gérer les tableaux est fourni.

##### Réponse :

```
1 #define _XOPEN_SOURCE 500
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <assert.h>
7
8 // note : la gestion mémoire est particulièrement brutale
9
10 // fonction qui sauvegarde un tableau de chaînes de caractères dans un
11 // fichier
12 void ecritTabChaines(const char *nomFichier, char * s[], int size)
13 {
14     FILE* f = fopen(nomFichier, "w+");
15
16     int s_len = size;
17     fwrite(&s_len, sizeof(int), 1, f);
18
19     for(int i = 0; i < size; i++)
20     {
21         s_len = strlen(s[i]) + 1;
22         fwrite(&s_len, sizeof(int), 1, f);
23         fwrite(s[i], sizeof(char), s_len, f);
24     }
25
26     fclose(f);
```

```

26 }
27
28 // fonction qui lit un tableau de chaînes de caractères
29 void litTabChaines(const char *nomFichier, char * tab[], int *size)
30 {
31     FILE* f = fopen(nomFichier, "r");
32     fread(size, sizeof(int), 1, f);
33
34     for(int i = 0; i < *size; i++)
35     {
36         int s_len = 0;
37         fread(&s_len, sizeof(int), 1, f);
38         fread(tab[i], sizeof(char), s_len, f);
39     }
40
41     fclose(f);
42 }
43
44
45 static int getNbChaine();
46 static char ** alloueInit();
47 static char ** alloue();
48 static void libere(char **tab);
49
50 int main()
51 {
52     int n = getNbChaine();
53     char **originaux = alloueInit();
54     char **copies = alloue();
55
56     int nbCopies;
57     ecritTabChaines("exoD_test", originaux, n);
58     litTabChaines("exoD_test", copies, &nbCopies);
59
60     if (nbCopies != n)
61         printf("PB ! mauvais nombre de chaînes lues\n");
62     else
63     {
64         for (int i = 0; i < nbCopies; i++)
65         {
66             if (strcmp(originaux[i], copies[i]) == 0)
67                 printf("ok, les deux chaînes numéro %2d sont identiques\n",
68 i);
69             else
70                 printf("PB ! Les deux chaînes numéro %2d diffèrent\n", i);
71         }
72     }
73
74     libere(originaux);
75     libere(copies);
76
77     return EXIT_SUCCESS;
78 }
```

### 12.5.5 Exercice E : tableaux d'entiers en binaire

**Question :** Le but ici est d'écrire un tableau d'entiers et de ne relire qu'un seul élément, le tout en mode binaire avec les fonctions fread et fwrite.

Écrire une fonction ;

```
void ecritTabIntBin(const char *nomFichier, const int tab[], int n)
```

qui écrit un tableau de int dans un fichier ; on ne met dans le fichier que les nombres du tableau et aucune autre information. L'écriture de tous les nombre doit être faite en une seule instruction.

Écrire une fonction ;

```
int litIntPosBin(const char *nomFichier, int pos)
```

qui lit un seul nombre (désigné par sa position) dans un fichier écrit par la fonction précédente.

Notes :

- Si l'indice est trop grand, on renvoie -1. On suppose qu'il n'y que des nombres positifs dans le fichier.
- On suppose aussi que la position passée en paramètre est positive.
- Pensez aux fonctions ftell et fseek.

Écrire une fonction ;

```
bool remplaceIntPosBin(const char *nomFichier, int pos, int newVal)
```

qui remplace un nombre (désigné par sa position) par un autre dans un fichier écrit par la fonction précédente. Si la position est trop grande, on ne fait rien et on renvoie false.

Dans la fonction main, faites le code qui teste les fonctions précédentes.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4
5
6 void ecritTabIntBin(const char *nomFichier, const int tab[], int n)
7 {
8     FILE* f = fopen(nomFichier, "w");
9     fwrite(tab, sizeof(int), n, f);
10    fclose(f);
11 }
12
13 int litIntPosBin(const char *nomFichier, int pos)
14 {
15     FILE* f = fopen(nomFichier, "r");
16
17     int f_size = 0;
18     fseek(f, 0, SEEK_END);
19     f_size = ftell(f);
20     fseek(f, pos * sizeof(int), SEEK_SET);
21
22     if(pos < (int)(f_size/sizeof(int)))
23     {
24         int res = 0;
25         fread(&res, sizeof(int), 1, f);
26     fclose(f);
27     return res;
28 }
29
30 else
```

```

31     {
32         fclose(f);
33         return -1;
34     }
35 }
36
37 bool remplaceIntPosBin(const char *nomFichier, int pos, int newVal)
38 {
39     FILE* f = fopen(nomFichier, "r+");
40     int oldVal = litIntPosBin(nomFichier, pos);
41     fseek(f, pos * sizeof(int), SEEK_SET);
42
43     if(oldVal < 0)
44     {
45         fclose(f);
46         return false;
47     }
48
49     else
50     {
51         fwrite(&newVal, sizeof(int), 1, f);
52         fclose(f);
53         return true;
54     }
55 }
56
57 int main()
58 {
59     int tab[] = {10, 5, 87, 0, 5, 212, 12, 14, 55};
60     int n = sizeof(tab)/sizeof(*tab);
61
62     // écrire le tableau dans le fichier exoE_test
63     ecritTabIntBin("exoE_test", tab, n);
64
65     // remplacer la case 2 (donc le 87) par 777
66     remplaceIntPosBin("exoE_test", 2, 777);
67     // remplacer la case 9 (indice trop grand) par 66 : il ne se passe rien
68     remplaceIntPosBin("exoE_test", 9, 66);
69     // remplacer la case 0 (donc le 10) par 1111
70     remplaceIntPosBin("exoE_test", 0, 1111);
71
72     // afficher le contenu du fichier en utilisant la fonction litIntPosBin
73     printf("[ ");
74
75     for (int i = 0; i < n; i++)
76     {
77         if (i != 0)
78             printf(", ");
79         printf("%d", litIntPosBin("exoE_test", i));
80     }
81
82     printf("]\n");
83
84     return EXIT_SUCCESS;
85 }
```

### 12.5.6 Exercice F : tableaux d'entiers en mode texte

Question :

Il s'agit donc du même exercice que le précédent, mais en mode texte, avec les fonctions `fprintf(..., "%d", ...)` et `fscanf(..., "%d", ...)`.

Écrire une fonction ;

`void ecritTabIntAsc(const char *nomFichier, const int tab[], int n)`

qui écrit un tableau de int dans un fichier ; on ne met dans le fichier que les nombres du tableau et aucune autre information : les nombres seront séparés par un espace.

Écrire une fonction ;

`int litIntPosAsc(const char *nomFichier, int pos)`

qui lit un seul nombre (désigné par sa position) dans un fichier écrit par la fonction précédente.

Notes :

- Si l'indice est trop grand, on renvoie -1. On suppose qu'il n'y que des nombres positifs dans le fichier.
- On suppose aussi que la position passée en paramètre est positive.

Écrire une fonction ;

`bool remplaceIntPosAsc(const char *nomFichier, int pos, int newVal)`

qui remplace un nombre (désigné par sa position) par un autre dans un fichier écrit par la fonction précédente. Si la position est trop grande, on ne fait rien et on renvoie false.

Attention c'est particulièrement pénible à faire ; ne faites cette fonction que lorsque le reste du TP est terminé.

Dans la fonction main, faites le code qui teste les fonctions précédentes.

Réponse :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4
5
6 void ecritTabIntAsc( const char *nomFichier , const int tab[] , int n)
7 {
8     FILE* f = fopen( nomFichier , "w" );
9
10    for( int i = 0; i < n; i++ )
11        fprintf(f , "%i " , tab[ i ] );
12
13    fclose( f );
14 }
15
16 int litIntPosAsc( const char *nomFichier , int pos )
17 {
18     FILE* f = fopen( nomFichier , "r" );
19
20     int res = 0;
21     int temp = 0;
22     int count = 0;
23
24     while( fscanf(f , "%i" , &temp) != EOF )
25     {
```

```

26     if (pos == count)
27         res = temp;
28
29     count++;
30 }
31
32 count--;
33
34 if (pos > count)
35 {
36     fclose(f);
37     return -1;
38 }
39
40 else
41 {
42     fclose(f);
43     return res;
44 }
45 }
46
47 bool remplaceIntPosAsc(const char *nomFichier, int pos, int newVal)
48 {
49     FILE* f = fopen(nomFichier, "r+");
50     FILE* ftemp = fopen("exoF_temp", "w+");
51     int oldVal = litIntPosAsc(nomFichier, pos);
52
53     int count = 0;
54     int temp = 0;
55
56     if (oldVal < 0)
57     {
58         printf("Hey!\n");
59         fclose(f);
60         return false;
61     }
62
63     else
64     {
65         while (fscanf(f, "%i ", &temp) != EOF)
66         {
67             int val = litIntPosAsc(nomFichier, count);
68
69             if (pos == count)
70             {
71                 fprintf(ftemp, "%i ", newVal);
72             }
73
74             else
75             {
76                 fprintf(ftemp, "%i ", val);
77             }
78
79             count++;
80         }
81     }

```

```

82 fclose( f );
83 remove( nomFichier );
84
85 fclose( ftemp );
86 rename( "exoF_temp" , nomFichier );
87
88 return true;
89 }
90 }
91
92 int main()
93 {
94     int tab[] = {10, 5, 87, 0, 5, 212, 12, 14, 55};
95     int n = sizeof(tab)/sizeof(*tab);
96     //printf("n is: %i\n", n);
97
98     // écrire le tableau dans le fichier exoF_test
99     ecritTabIntAsc( "exoF_test" , tab , n );
100
101    // remplacer la case 2 (donc le 87) par 9
102    remplaceIntPosAsc( "exoF_test" , 2 , 9 );
103    // remplacer la case 9 (indice trop grand) par 66 : il ne se passe rien
104    remplaceIntPosAsc( "exoF_test" , 9 , 66 );
105    // remplacer la case 0 (donc le 10) par 1111
106    remplaceIntPosAsc( "exoF_test" , 0 , 1111 );
107
108    // afficher le contenu du fichier en utilisant la fonction litIntPosAsc
109    printf( " [ " );
110    for ( int i = 0; i < n; i++ )
111    {
112        if ( i != 0 )
113            printf( ", " );
114        printf( "%d" , litIntPosAsc( "exoF_test" , i ) );
115    }
116    printf( " ]\n" );
117
118    return EXIT_SUCCESS;
119 }
```

### 12.5.7 Exercice G : Lecture / écriture d'une structure

Question :

Soit le type suivant :

```
1 struct S
2 {
3     int i;
4     char c;
5     float f;
6     int tab[10];
7 };
```

Écrire une fonction ;

*void ecritStructBin(const char \*nomFichier, const struct S \*s)*

qui écrit, en binaire, une structure dans un fichier ; l'écriture doit être faite en une seule instruction.

Écrire une fonction ;

*void litStructBin(const char \*nomFichier, struct S \*s)*

qui lit une structure dans un fichier créé par la fonction précédente ; la lecture doit être faite en une seule instruction.

Écrire une fonction ;

*void tailleStruct()*

qui affiche la taille totale de la structure, puis la somme des tailles des 4 champs.

Dans la fonction main, faites le code qui teste les fonctions précédentes.

Réponse :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h> // Nécessaire pour memcmp
4
5 struct S
6 {
7     int i;
8     char c;
9     float f;
10    int tab[10];
11 };
12
13 void ecritStructBin(const char *nomFichier, const struct S *s)
14 {
15     FILE* f = fopen(nomFichier, "w");
16     fwrite(s, sizeof(const struct S), 1, f);
17     fclose(f);
18 }
19
20 void litStructBin(const char *nomFichier, struct S *s)
21 {
22     FILE* f = fopen(nomFichier, "r");
23     fread(s, sizeof(struct S), 1, f);
24     fclose(f);
25 }
```

```

27 void tailleStruct()
28 {
29     struct S x = {0, 'o', 0, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}};
30     printf("struct S: %ld\n\
31         Field int i : %ld\n\
32         Field char c: %ld\n\
33         Field float f: %ld\n\
34         Field int tab[10]: %ld\n",
35         sizeof(x), sizeof(x.i),
36         sizeof(x.c), sizeof(x.f),
37         sizeof(x.tab));
38 }
39
40 int main()
41 {
42     struct S original = {42, 'a', 3.14159, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}};
43     struct S copie;
44
45     // écrire la structure "original" dans le fichier exoG_test
46     // todo
47     écritStructBin("exoG_test", &original);
48
49     // lire le fichier et remplir la structure "copie"
50     // todo
51     litStructBin("exoG_test", &copie);
52
53     // comparer les deux structures
54     // todo
55     if (memcmp(&original, &copie, sizeof(struct S)) == 0)
56         printf("Ok les structures sont identiques\n");
57     else
58         printf("PB ! Les structures sont différentes\n");
59
60     // appeler la fonction tailleStruct
61     // todo
62     tailleStruct();
63
64     return EXIT_SUCCESS;
65 }
```

**Question :** Lorsqu'on compare la taille de la structure, la somme des tailles des 4 champs et la taille du fichier, a-t-on les mêmes nombres ? Si ce n'est pas le cas, quelles sont les données en trop ou en moins ?

**Réponse :** La taille de la structure (52), du fichier (52 d'après la commande ls -lh dans Bash) et la somme des tailles des 4 champs (49) ne sont pas égales. En effet, une structure semble être calculée à partir de son premier champ, ici un int qui prend 4 octets. Or  $4 * 12 = 48$  octets et on a besoin de 49 octets. Donc  $4 * 13 = 52$  octets et on a 3 octets inutiles mais la place nécessaire...

Comme on écrit avec des fonctions d'écriture binaire on écrit aussi ces 3 octets inutiles, d'où la taille du fichier valant 52 octets.

### 12.5.8 Exercice H

**Question :**

Avec open, dprintf et close faire le programme suivant :

- créer le fichier exoH test
- écrire dedans la chaîne "AA\n"
- écrire dedans la chaîne "BB\n"
- écrire dedans la chaîne "CC\n"
- attendre 10 secondes (fonction sleep)
- écrire dedans la chaîne "DD\n"
- fermer le fichier

Le but est d'interrompre violemment le programme avec Ctrl-C lors de l'attente et de regarder le contenu du fichier partiellement rempli.

**Réponse :**

```
1 #define _XOPEN_SOURCE 700 // pour dprintf
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6 #include <sys/types.h>
7 #include <sys/stat.h>
8 #include <fcntl.h>
9
10
11 int main()
12 {
13     int fd = open("./exoH_1_test", O_CREAT | O_WRONLY, 00644);
14     dprintf(fd, "%s", "AA\n");
15     dprintf(fd, "%s", "BB\n");
16     dprintf(fd, "%s", "CC\n");
17
18     sleep(10);
19     dprintf(fd, "%s", "DD\n");
20
21     close(fd);
22
23     return EXIT_SUCCESS;
24 }
```

**Question :** Est-ce le résultat attendu ?

**Réponse :** Oui, dprintf est une fonction d'écriture non bufferisée, elle écrit donc dès qu'elle est appelée

### 12.5.9 Exercice I

**Question :**

Il s'agit exactement du même exercice mais avec les fonctions fopen, fprintf et fclose et le fichier exoI test.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int main()
{
6     FILE* f = fopen("./exoI_test", "w");
7     fprintf(f, "%s", "AA\n");
8     fprintf(f, "%s", "BB\n");
9     fprintf(f, "%s", "CC\n");
10    sleep(10);
11    fprintf(f, "%s", "DD\n");
12
13    fclose(f);
14
15    return EXIT_SUCCESS;
16}
17
18}
```

**Question :** A-t-on le même comportement que l'exercice précédent ? Comment l'expliquez-vous ?  
(c'est une question d'entrées-sorties bufferisées)

**Réponse :** La fonction fprintf est une fonction d'écriture bufferisée. Ce qui signifie que les chaînes de caractères "AA\n", etc. sont stockées dans la mémoire vive en attendant la fin du programme ou un fflush (ou qu'elle soit pleine?). Comme on interrompt le programme avant le return EXIT\_SUCCESS les chaînes ne sont pas écrites dans le fichier contrairement aux fonctions d'écritures non bufferisées précédentes qui écrivent dès que la fonction d'écriture est appelée.

# Chapitre 13

## Abstraction pointeur

### Exercice 1

**Question :** Le but est de montrer que l'on peut écrire un programme (main.c) utilisant des bibliothèques (Voiture et Collection) sans connaître leurs implémentations 2 et même sans que ces dernières ne soient écrites. En revanche il faut connaître leurs interfaces (i.e. les .h).

Écrire uniquement les fichiers suivants (à l'exclusion de tout autre) :

- Voiture.h entièrement (avec l'abstraction pointeur)
- Collection.h entièrement (avec l'abstraction pointeur)
- main.c

La fonction main devrait contenir des appels à toutes les méthodes déclarées dans les .h pour faire une couverture de tests complète. Dans cet exercice il vous est seulement demandé de choisir 5 méthodes de chaque module et les appeler.

Le fichier main.c doit compiler sans erreur ni warning :

```
$ make main.o
```

ou

```
$ gcc -g -Wall -Wextra -pedantic -std=c99 -c main.c
```

Comme les deux modules ne sont pas implémentés, il possible de compiler le fichier main.c, mais pas de générer l'exécutable (édition de liens).

Sont fournis les fichiers suivants :

- Makefile
- myassert.h

Le fichier myassert.h n'est a priori pas à utiliser dans cette partie

**Réponse :**

```
1 #ifndef COLLECTION_H
2 #define COLLECTION_H
3
4 struct carCollectionP;
5 typedef struct carCollectionP* carCollection;
6 typedef const struct carCollectionP* const_carCollection;
7
8 carCollection createEmptyCarCollection();
9 carCollection createCopiedCarCollection(const_carCollection c);
```

```

10
11 void destroyCarCollection(carCollection c);
12 void emptyCarCollection(carCollection c);
13
14 int getCarCollectionSize(const_carCollection c);
15 car getCar(const_carCollection c, int index);
16
17 void unsortedAddCar(carCollection coll, const_car c);
18 void sortedAddCar(carCollection coll, const_car c);
19 void unsortedRemoveCar(carCollection c, int index);
20 void sortedRemoveCar(carCollection c, int index);
21
22 void sortCarCollection(carCollection c);
23
24 void printCarCollection(const_carCollection c);
25 void saveCarCollection(const_carCollection c, FILE *f);
26 void reinitCarCollection(carCollection c, FILE *f);
27
28 #endif

```

**Listing 13.1** – Collection.h

```

1 #ifndef VOITURE_H
2 #define VOITURE_H
3
4
5 struct carP;
6 typedef struct carP* car;
7 typedef const struct carP* const_car;
8
9 car createDefinedCar(char brand[], int year, int km, char** immat, int
10 nbImmat);
11 car createCopiedCar(const_car c);
12 car createSavedCar(const char *nomFichier);
13 void switchCarData(car c1, car c2);
14 void destroyCar(car c);
15
16 char* getBrand(const_car c);
17 int getYear(const_car c);
18 int getKm(const_car c);
19 void setKm(car c, int km);
20 int getNbImmat(const_car c);
21 int getImmat(const_car c, int index);
22 int addImmat(car c, char* immat);
23
24 void printCar(const_car c);
25 void saveCar(const_car c, FILE *f);
26 void copyFileCar(car c, FILE *f);
27
28 int getNbInitCar();
29 int getNbInitImmat();
30 int getMinYear();
31 int getMaxYear();
32 #endif

```

**Listing 13.2** – Voiture.h

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include "Voiture.h"
5 #include "Collection.h"
6
7 int main( int argv , char *argc [] )
8 {
9
10    if( argv > 1 && !strcmp( argc[1] , "full" ) )
11    {
12    }
13
14    else
15    {
16        /* On teste 5 méthodes du module Voiture.h */
17
18        char* immatList [] = {"FLORIAN73"};
19
20        //1
21        car myCar = createDefinedCar("Toyota" , 2023 , 0 , immatList , 1);
22
23        //2
24        printCar(myCar);
25
26        //3
27        FILE *f = fopen("./mycar" , "w+");
28        saveCar(myCar , f);
29        fclose(f);
30
31        //4
32        car mySavedCar = createSavedCar("./mycar");
33        printCar(mySavedCar);
34
35        //5
36        car myCopiedCar = createCopiedCar(myCar);
37        printCar(myCopiedCar);
38
39        /* On teste 5 méthodes du module Collection.h */
40
41        //1
42        carCollection myColl = createEmptyCarCollection();
43
44        //2
45        sortedAddCar( myColl , myCar );
46
47        //3
48        carCollection myCopiedColl = createCopiedCarCollection( myColl );
49
50        //4
51        printCarCollection( myColl );
52        printCarCollection( myCopiedColl );
53
54        //5
55        destroyCarCollection( myColl );

```

```
56     destroyCarCollection (myCopiedColl) ;  
57  
58 // Destruction des voitures  
59 destroyCar (myCar) ;  
60 destroyCar (myCopiedCar) ;  
61 destroyCar (mySavedCar) ;  
62  
63     return EXIT_SUCCESS;  
64 }  
65 }
```

**Listing 13.3** – main.c

## Exercice 2

**Question :** On oublie les 3 fichiers précédents, de nouvelles versions sont fournies et imposées (i.e. obligation de les utiliser et interdiction de les modifier par la suite).

En outre les versions compilées de Voiture.c et Collection.c sont également fournies (en espérant qu'elles sont compatibles avec votre architecture).

Pour récapituler, voici la liste des fichiers disponibles :

```
— Makefile
— myassert.h
— myassert.c
— main.c
— Voiture.h
— Voiture.o
— Collection.h
— Collection.o
```

Le seul et unique but de cet exercice est de vérifier que l'ensemble compile :

```
$ make
```

ou

```
$ gcc -g -Wall -Wextra -pedantic -std=c99 myassert.c main.c Collection.o
      Voiture.o -o main
```

et de lancer l'exécution :

```
$ ./main
```

Si vous passez plus de 60 secondes sur cet exercice, cela signifie que vous faites plus que ce qui est demandé (comme écrire du code par exemple).

**Réponse :** On obtient un affichage que nous chercherons à reproduire dans les exercices 3 et 4 suivant... En voici un extrait :

```
=====
= Statistiques
=====

Nb initialisations : 0
Nb immatriculations : 0
Année minimale      : -1
Année maximale      : -1

=====
= Voitures
=====

v1 :
v1 a pour marque Trombine (Trombine normalement)
v1 a pour année 2005 (2005 normalement)
v1 a pour kilométrage 38951 (38951 normalement)
v1 a 3 immatriculations (3 normalement)
[1] 1234 AE 75
[2] VH 529 FE
[3] VY 749 RT
```

**Listing 13.4** – Extrait de l'exécution du main dans Bash

## Exercice 3

**Question :** Écrire uniquement le fichier Collection.c pour générer votre propre fichier Collection.o. Comme le module Collection utilise Voiture, le fichier Voiture.o (i.e. la version compilée de Voiture.c) est fourni.

Les voitures sont stockées dans un tableau alloué dynamiquement (on impose que ce soit un tableau). Le nombre de cases du tableau correspond exactement au nombre de voitures stockées.

Compiler avec le main.c et le Voiture.o fournis, et vérifier que tout fonctionne :

```
$ make
```

ou

```
$ gcc -g -Wall -Wextra -pedantic -std=c99 myassert.c main.c Collection.c  
Voiture.o -o main
```

Rappel : interdit de modifier main.c, Collection.h ou Voiture.h. Si vous êtes tentés de les modifier, alors vous ne répondez pas à la question.

Rappel : Collection.c n'a pas le droit de manipuler directement les champs de la structure Voiture, mais juste le droit d'appeler les méthodes de Voiture.h (et si Collection.c arrive à manipuler directement la structure de Voiture je suis intéressé de savoir comment vous avez fait).

Sont fournis les fichiers suivants :

- Makefile
- myassert.h
- myassert.c
- main.c
- Voiture.h
- Voiture.o
- Collection.h

**Réponse :**

```
1 struct CollectionP  
2 {  
3     Voiture* voiTab;  
4     int voiTabSize;  
5     bool isSorted;  
6 };
```

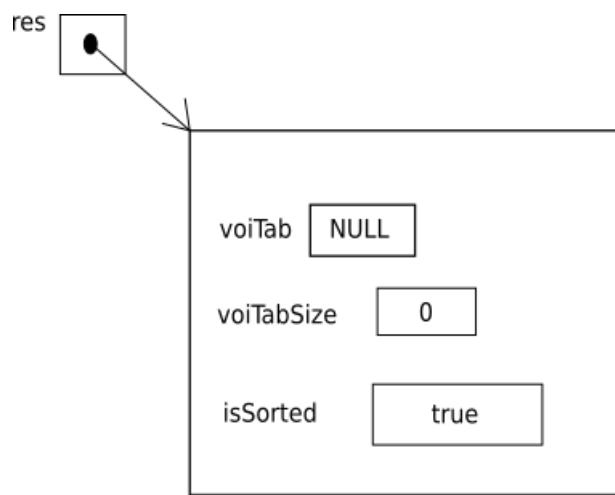
**Listing 13.5** – Implémentation de la structure cachée dans le .h

On note ici qu'on rajoute la taille du tableau alors que celle-ci n'est pas indiquée dans le cahier des charges pour pouvoir itérer sur les éléments de la collection.

```

1 Collection col_creer()
2 {
3     Collection res = (Collection) malloc(sizeof(struct CollectionP));
4     res->voiTab = NULL;
5     res->voiTabSize = 0;
6     res->isSorted = true;
7
8     return res;
9 }
```

**Listing 13.6** – Implémentation de la fonction de création d'une collection vide indiquée dans le .h



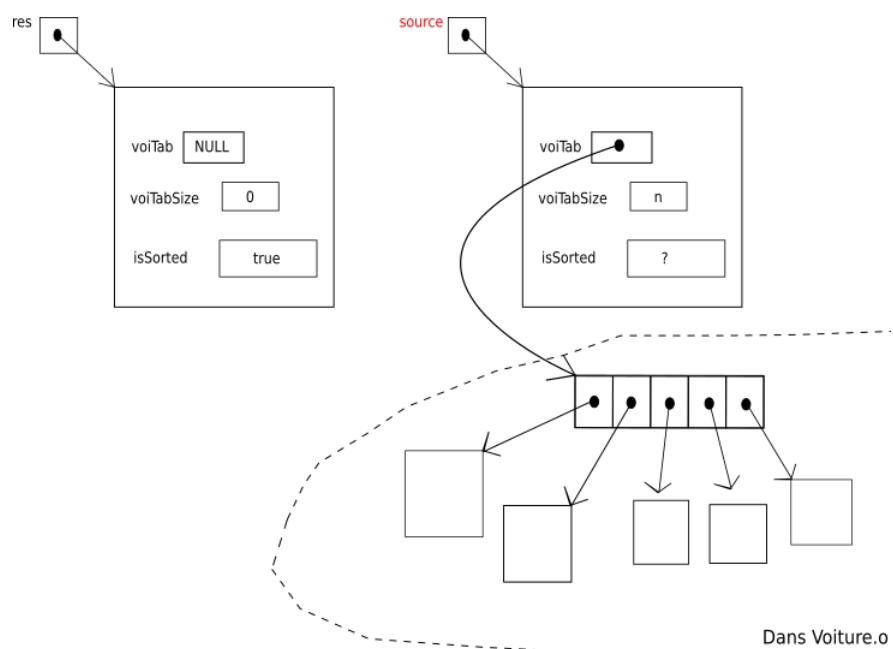
**FIGURE 13.1** – Illustration de ce qui est cree par la fonction `col_creer()`

```

1 Collection col_creerCopie(const_Collection source)
2 {
3     Collection res = col_creer();
4     res->voiTab = (Voiture*) realloc(res->voiTab, source->voiTabSize *
5         sizeof(Voiture));
6     for (int i = 0; i < source->voiTabSize; i++)
7         res->voiTab[i] = voi_creerCopie(source->voiTab[i]);
8
9     res->voiTabSize = source->voiTabSize;
10    res->isSorted = source->isSorted;
11
12    return res;
13}

```

**Listing 13.7** – Implémentation de la fonction de copie d'une collection indiquée dans le .h



**FIGURE 13.2** – Illustration de la situation dans la fonction `col_creerCopie()` à la ligne 3

On a la situation illustrée ci-dessus. Notre source est en rouge car elle est constante (on ne peut pas écrire dedans mais on peut lire ses champs). Dans les pointillés sont indiqués ce qu'il se passe technique-ment dans Voiture.c (ou le .o) mais bien sûr, nous qui implantons le module de collection nous ne savons pas si c'est vraiment ce qu'il se passe. Tout ce que nous savons c'est que `voiTab` pointe vers le premier élément d'un tableau de voitures mais nous ne savons pas que les voitures sont des pointeurs...

Notre premier objectif est d'allouer de l'espace à notre tableau `voiTab`. Cet espace doit être de taille suffisante, ie. qu'elle doit être de la taille du tableau qu'on souhaite copier : `source->voiTab`. Cette taille, on la connaît, elle est dans le tableau `source` : `source->voiTabSize`. D'où la ligne 4.

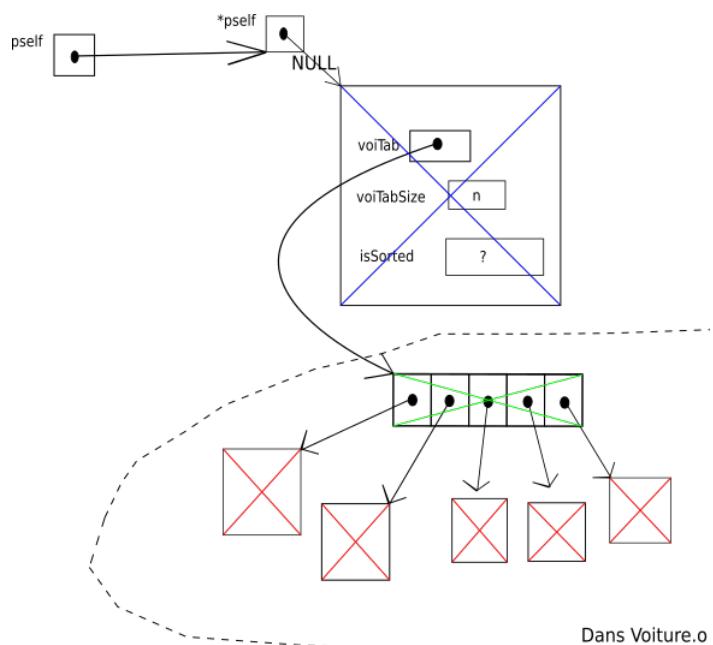
On peut alors créer des copies de voitures à chaque case de notre nouveau tableau en utilisant la fonction de copie de voiture fournie par `Voiture.h`.

Les autres copies sont directes (il s'agit de simples valeurs) et ne nécessitent pas d'allocations ou de méthodes particulières.

```

1 void col_detruire(Collection* pself)
2 {
3     for(int i = 0; i < (*pself)->voiTabSize; i++)
4         voi_detruire(&((*pself)->voiTab[i]));
5
6     free((*pself)->voiTab);
7     free(*pself); //Libère la zone pointée par *pself...
8     *pself = NULL; //... mais *pself continue de pointer vers la zone! D'où
9 }
```

**Listing 13.8** – Implémentation de la fonction de destruction d'une collection indiquée dans le .h



**FIGURE 13.3** – Illustration de la situation dans la fonction `col_creerCopie()` à la ligne 3

Si on commençait la désallocation mémoire (ie. la destruction) par le haut (ie. `*pself`), nous n'aurions plus accès à toute la chaîne de pointage qui commence par `*pself`. Or cette chaîne resterait allouée car nous n'aurions pas appelé la fonction `free()` sur ces zones !

On commence donc par le bas (les croix rouges) en itérant parmi les voitures. Une fois que c'est fait, on désalloue la zone que nous avions allouée à notre tableau (la croix verte) et ensuite la zone pointée par `*pself` (la croix bleue).

Enfin, on note : `free(*ptr)` libère la zone pointée par pointeur, mais elle ne change pas la valeur (l'adresse de début de la zone allouée) de `ptr` qui pointe alors vers une zone mémoire qui peut maintenant appartenir à un autre processus (allocation => la zone appartient à notre processus tant qu'on le veut. Déallocation => la zone peut appartenir à n'importe quel processus du système.) On n'oubliera donc pas d'assigner la valeur de `*pself` à `NULL`. C'est pour ça qu'on passe un pointeur vers une collection... Si on ne faisait pas ça on assignerait à `NULL` une copie de ce pointeur ce qui ferait que notre collection pointerait toujours vers la zone qui ne nous appartient plus.

```

1 void col_vider(Collection self)
2 {
3     for(int i = 0; i < self->voiTabSize; i++)
4         voi_detruire(&(self->voiTab[i]));
5
6     self->voiTabSize = 0;
7     self->isSorted = true;
8 }
```

**Listing 13.9** – Implémentation de la fonction de vidage d'une collection indiquée dans le .h

La fonction est très similaire à la fonction de destruction. On note toutefois qu'on ne désalloue pas la zone pointée par voiTab. Nous pourrions le faire pour recréer une structure exactement comme celle générée par col\_creer(). Ici, je ne l'ai pas fait car vider une collection signifie qu'on se resservira de celle-ci donc j'ai préféré laisser une zone allouée quitte à la réallouer par la suite (si les collections ont des tailles très importantes il faudrait désallouer cette zone inutilisée !)

```

1 Voiture col_getVoiture(const.Collection self, int pos)
2 {
3     myassert_func(pos >=0, "Invalid pos, pos < 0",
4                 "Collection.c", "col_getVoiture", 1);
5     myassert_func(pos < self->voiTabSize, "Invalid pos, pos > self->
6                 voiTabSize",
7                 "Collection.c", "col_getVoiture", 2);
8
9     Voiture res = voi_creerCopie(self->voiTab[pos]);
10    return res;
}
```

**Listing 13.10** – Implémentation de la fonction de récupération d'une voiture comme indiquée dans le .h

On note ici l'utilisation de la bibliothèque myassert.h pour détecter des erreurs d'utilisation de la fonction col\_getVoiture().

```

1 void col_addVoitureSansTri(Collection self, const.Voiture voiture)
2 {
3     self->voiTabSize++;
4     self->voiTab = (Voiture*) realloc(self->voiTab,
5                                         self->voiTabSize * sizeof(Voiture));
6     self->voiTab[self->voiTabSize - 1] = voi_creerCopie(voiture);
7
8     if (self->voiTabSize > 1)
9         self->isSorted = false;
10 }
```

**Listing 13.11** – Implémentation de la fonction d'ajout d'une voiture dans la collection sans tri comme indiquée dans le .h

On note simplement l'ajout en queue de la voiture dans la collection pour une insertion en complexité constante. On pourrait rajouter un myassert() pour vérifier si la collection est triée ou non afin d'empêcher qu'on n'ôte le tri d'une collection par une insertion sans tri.

```

1 void col_addVoitureAvecTri( Collection self , const_Voiture voiture)
2 {
3     //On teste le paramètre d'entrée et on fait les initialisations
4     //qui peuvent être faites directement
5     myassert_func(self->isSorted , "Collection not sorted",
6                 "Collection.c" , "col_addVoitureAvecTri" , 1);
7
8     self->voiTabSize++;
9     self->voiTab = ( Voiture* ) realloc( self->voiTab ,
10                                         self->voiTabSize * sizeof( Voiture ) );
11
12    //Recherche dichotomique de la position à laquelle on doit
13    //ajouter la nouvelle voiture (donc en log2(n))
14    int lower_bound = 0;
15    int higher_bound = self->voiTabSize -1;
16    int median = (lower_bound + higher_bound) / 2;
17    int range = self->voiTabSize ;
18    int year = voi_getAnnee( voiture );
19    int tmp_year = 0;
20    int res_pos = 0;
21
22    while( range > 1 )
23    {
24        tmp_year = voi_getAnnee( self->voiTab [ median ] );
25        range = higher_bound - lower_bound ;
26
27        if( year <= tmp_year )
28        {
29            higher_bound = median ;
30        }
31
32        else
33        {
34            lower_bound = median ;
35        }
36
37        median = (lower_bound + higher_bound) / 2;
38    }
39
40    if( year <= tmp_year )
41        res_pos = median ;
42
43    else
44        res_pos = median + 1;
45
46    //Déplacement des éléments du tableau pour laisser la place à la
47    //nouvelle voiture... On choisit memmove car les zones src et dest
48    //sont allouées et se chevauchent
49    if( res_pos != 0 )
50    {
51        memmove( &( self->voiTab [ res_pos ] ) , &( self->voiTab [ res_pos -1 ] ) ,
52                ( self->voiTabSize - res_pos ) * sizeof( Voiture ) );
53    }
54
55    else

```

```

56 {
57     memmove(&(self->voiTab[1]), &(self->voiTab[0]),
58             (self->voiTabSize - 1 - res_pos) * sizeof(Voiture));
59 }
60
61 //Ajout de la nouvelle voiture
62 self->voiTab[res_pos] = voi_creerCopie(voiture);
63
64
65 //Fonctionne mais complexité quadratique due à col_trier:
66 //col_addVoitureSansTri(self, voiture);
67 //col_trier(self);
68
69 }

```

**Listing 13.12** – Implémentation de la fonction d'ajout d'une voiture dans la collection avec tri comme indiquée dans le .h

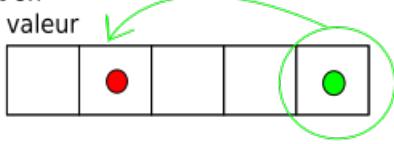
Une recherche dichotomique dans le tableau de voiture est implémentée directement dans la fonction bien qu'elle puisse être implantée par une fonction static afin d'alléger la lecture de cette fonction tout en la rendant locale au fichier Collection.c.

```

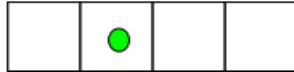
1 void col_supprVoitureSansTri(Collection self, int pos)
2 {
3     myassert_func(pos >= 0, "Invalid pos, pos < 0", "Collection.c",
4                 "col_supprVoitureSansTri", 1);
5     myassert_func(pos < self->voiTabSize, "Invalid pos, pos > self->
6                 voiTabSize",
7                 "Collection.c", "col_supprVoitureSansTri", 2);
8
9     voi_detruire(&(self->voiTab[pos]));
10    self->voiTab[pos] = self->voiTab[self->voiTabSize - 1];
11    self->voiTabSize--;
12    self->voiTab = (Voiture*) realloc(self->voiTab,
13                                       self->voiTabSize * sizeof(Voiture));
14
15 //Fonctionne mais complexité linéaire:
16 /*
17 for( ; pos < self->voiTabSize - 1; pos++)
18 {
19     voi_swap(self->voiTab[pos], self->voiTab[pos+1]);
20 }
21
22 voi_detruire(&(self->voiTab[pos]));
23 self->voiTabSize--;
24 self->voiTab = (Voiture*) realloc(self->voiTab,
25                                     self->voiTabSize * sizeof(Voiture));
26 */
}
```

**Listing 13.13** – Implémentation de la fonction de suppression d'une voiture dans la collection sans tri comme indiquée dans le .h

On désalloue la zone pointée par le pointeur rouge et on affecte à ce pointeur la valeur du pointeur vert:



Après la réallocation:



**FIGURE 13.4** – Illustration de la situation dans la fonction col\_creerCopie() à la ligne 3

L'illustration ci-dessus explique les instructions effectuées ligne 8 à 13 du listing précédent. En effet, avec voi\_detruire on détruit la zone allouée pointée par la case du tableau &(self->voiTab[pos]). Ensuite, on affecte à la valeur située dans pos la valeur en bout de tableau. Enfin, on réalloue l'espace par le tableau (on diminue la zone allouée) en indiquant dans le champ voiTabSize que le tableau a diminué d'un élément.

```

1 void col_supprVoitureAvecTri(Collection self, int pos)
2 {
3     myassert_func(pos >=0, "Invalid pos, pos < 0", "Collection.c",
4                 "col_supprVoitureAvecTri", 1);
5     myassert_func(pos < self->voiTabSize, "Invalid pos, pos > self->
6                 voiTabSize",
7                 "Collection.c", "col_supprVoitureAvecTri", 2);
8
9     for( ; pos < self->voiTabSize - 1; pos++)
10    {
11        voi_swap(self->voiTab[pos], self->voiTab[pos+1]);
12    }
13
14    voi_detruire(&(self->voiTab[pos]));
15    self->voiTabSize--;
16    self->voiTab = (Voiture*) realloc(self->voiTab,
17                                       self->voiTabSize * sizeof(Voiture));
18
19 //Fonctionne mais complexité quadratique due à col_trier:
20 //col_supprVoitureSansTri(self, pos);
21 //col_trier(self);
}

```

**Listing 13.14** – Implémentation de la fonction de suppression d'une voiture dans la collection avec tri comme indiquée dans le .h

On note ici que la fonction conserve certes le tri mais a une complexité linéaire (on swap l'élément tout au bout du tableau). On pourrait utiliser la fonction memmove() pour proposer une suppression en temps constant.

```

1 void col_ecrireFichier(const_Collection self, FILE *fd)
2 {
3     fwrite(&(self->voiTabSize), sizeof(int), 1, fd);
4     fwrite(&(self->isSorted), sizeof(bool), 1, fd);
5
6     for(int i = 0; i < self->voiTabSize; i++)
7         voi_ecrireFichier(self->voiTab[i], fd);
8 }
9
10 void col_lireFichier(Collection self, FILE *fd)
11 {
12     col_vider(self);
13     fread(&(self->voiTabSize), sizeof(int), 1, fd);
14     fread(&(self->isSorted), sizeof(bool), 1, fd);
15     self->voiTab = (Voiture*) malloc(self->voiTabSize * sizeof(Voiture));
16
17     for(int i = 0; i < self->voiTabSize; i++)
18         self->voiTab[i] = voi_creerFromFichier(fd);
19 }
```

**Listing 13.15** – Fonction de suppression d'une voiture dans la collection sans tri comme indiquée dans le .h

On évite un simple fwrite() et fread() de la collection elle-même ou encore un fwrite() ou fread() de voiTab. En effet, dans le premier cas on écrirait alors en binaire dans le fichier passé en paramètre la valeur du pointeur de la collection. Ce qui signifie que si cette collection est par la suite détruite et qu'on utilise col\_lireFichier() alors on aura une collection dont la valeur est un pointeur vers une zone qui ne lui appartient pas. Même remarque dans le second cas mais avec voiTab. On écrit donc et on lira des valeurs de champs et non des adresses de zones qui peuvent être désallouée !

```

1 void col_trier(Collection self)
2 {
3     for(int i = 0; i < self->voiTabSize; i++)
4     {
5         for(int j = 0; j < self->voiTabSize - i - 1; j++)
6         {
7             int year1 = voi_getAnnee(self->voiTab[j]);
8             int year2 = voi_getAnnee(self->voiTab[j+1]);
9
10            if(year1 > year2)
11            {
12                voi_swap(self->voiTab[j], self->voiTab[j+1]);
13            }
14        }
15    }
16
17    self->isSorted = true;
18 }
19
20 }
```

```

21 /*—————*
22 * méthode secondaire d'affichage
23 *—————*/
24
25 void col_afficher( const_Collection self )
26 {
27     for( int i = 0; i < self->voiTabSize; i++ )
28         voi_afficher( self->voiTab[ i ] );
29 }
```

**Listing 13.16** – Implémentation de la fonction de suppression d'une voiture dans la collection sans tri comme indiquée dans le .h

Pour la fonction de tri on utilise un algorithme de type bubble\_sort(). L'affichage d'une collection utilise directement voi\_afficher() de Voiture.h.

## Exercice 4

### 13.4.1 Implémentation de Voiture.c

**Question :** Écrire le fichier Voiture.c pour générer votre propre fichier Voiture.o Les chaînes de caractères sont allouées dynamiquement et aux tailles strictement nécessaires (pas de surdimensionnement).

Il en va de même pour le tableau des immatriculations (on impose que les immatriculations soient stockées dans un tableau).

Testez votre code avec les deux compilations suivantes :

- avec le fichier Collection.o fourni et votre Voiture.c (pour se concentrer sur le débogage du module Voiture)
- avec vos deux fichiers Voiture.c et Collection.c

Rappel : dans cet exercice, seul le fichier Voiture.c doit être écrit et il est interdit de modifier main.c, Collection.h ou Voiture.h

Sont fournis les fichiers suivants :

- Makefile
- myassert.h
- myassert.c
- main.c
- Voiture.h
- Collection.h

**Réponse :**

```
1 #define _XOPEN_SOURCE 500 //Pour strdup
2 //Doit être placé avant le include de string.h
3 #include <stdlib.h>
4 #include <string.h>
5 #include <stdbool.h>
6 #include "Voiture.h"
7
8 #define MAX_LEN 1000
9
10 struct VoitureP
11 {
12     char* brand;
13     int year;
14     int km;
15     char** immats;
16     int nbImmat;
17 };
```

**Listing 13.17** – Implémentation de la structure VoitureP cachée par l'abstraction pointeur dans Voiture.h

```

1 static int nbInit = 0;
2 static int nbImmat = 0;
3 static int minYear = -1;
4 static int maxYear = -1;
5
6 int voi_stat_getNbInitialisations()
7 {
8     return nbInit;
9 }
10
11 int voi_stat_getNbImmatriculations()
12 {
13     return nbImmat;
14 }
15
16 int voi_stat_getAnneeMin()
17 {
18     return minYear;
19 }
20
21 int voi_stat_getAnneeMax()
22 {
23     return maxYear;
24 }
```

**Listing 13.18** – Implémentation des fonctions de statistiques partie 1

```

1 static void stats_tracker(int annee)
2 {
3     nbInit++;
4     static bool first_time = true;
5
6     if(first_time)
7     {
8         minYear = annee;
9         maxYear = annee;
10        first_time = false;
11    }
12
13    else
14    {
15        if(annee < minYear)
16            minYear = annee;
17
18        else if(annee > maxYear)
19            maxYear = annee;
20    }
21 }
```

**Listing 13.19** – Implémentation des fonctions de statistiques partie 2

La seule statistique qui n'est pas suivie ("trackée") par stats\_tracker() est nbImmat qui suivie dans la fonction d'ajout d'une immatriculation.

```

1 Voiture voi_creer(const char* marque, int annee, int kilometrage, int
2 nbImmatriculations, const char* immatriculations[])
{
3     stats_tracker(annee);
4
5     Voiture res = (Voiture) malloc(sizeof(struct VoitureP));
6     char* myBrand = strdup(marque);
7     res->brand = myBrand;
8     res->brand = strcpy(res->brand, marque);
9     res->year = annee;
10    res->km = kilometrage;
11    res->nbImmat = 0;
12
13    char** immaTab = (char **) malloc(res->nbImmat * sizeof(char *));
14    res->immats = immaTab;
15
16    for(int i = 0; i < nbImmatriculations; i++)
17    {
18        voi_addImmatriculation(res, immatriculations[i]);
19    }
20
21    return res;
22}

```

**Listing 13.20** – Fonction de création de la voiture avec des paramètres correspondant à l’ensemble des champs du type

```

1 Voiture voi_creerCopie(const_Voiture source)
2 {
3     Voiture res = voi_creer(source->brand, source->year, source->km,
4                             source->nbImmat, (const char**) source->immats);
5     return res;
6 }

```

**Listing 13.21** – Fonction de copie de voiture qui réutilise la fonction de création précédente

```

1 Voiture voi_creerFromFichier(FILE *fd)
2 {
3     Voiture res = (Voiture) malloc(sizeof(struct VoitureP));
4     res->brand = NULL; //On évite une erreur de variable non initialisée
5         //de valgrind...
6     res->immats = NULL; //Même chose ici!
7
8     voi_lireFichier(res, fd);
9
10    return res;
11 }

```

**Listing 13.22** – Crédit d’une voiture par la lecture d’un fichier où on a sauvegardé (écrit) les champs d’une autre voiture

```

1 void voi_detruire(Voiture *pself)
2 {
3     for (int i = 0; i < (*pself)->nbImmat; i++)
4     {
5         free ((*pself)->immats[i]);
6     }
7
8     free ((*pself)->immats);
9     free ((*pself)->brand);
10    free (*pself);
11    *pself = NULL;
12 }
```

**Listing 13.23** – Destruction d'une voiture

Comme pour la fonction de destruction d'une collection, on part du bas de la chaîne de pointage débutant en `*pself` puis on remonte à `*pself`. Pour les mêmes raisons détaillées dans l'exercice 3 à la fonction de destruction d'une collection, ici on passe une voiture par référence afin d'éviter que notre voiture du main.c continue de pointer vers une zone mémoire qui ne lui appartient plus.

```

1 void voi_setKilometrage(Voiture self, int kilometrage)
2 {
3     if (self->km > kilometrage)
4         exit(EXIT_FAILURE);
5
6     else
7         self->km = kilometrage;
8 }
```

**Listing 13.24** – Mutateur du champ de kilométrage d'une voiture

On note ici que nous aurions pu utiliser un assert avec un message personnalisé pour éviter un arrêt brutal du programme sans information sur l'erreur détectée.

```

1 void voi_addImmatriculation(Voiture self, const char* immatriculation)
2 {
3     nbImmat++;
4     self->nbImmat++;
5     self->immats = (char **) realloc(self->immats, self->nbImmat * sizeof(
6         char *));
7     self->immats[self->nbImmat - 1] = strdup(immatriculation);
}
```

**Listing 13.25** – Ajout d'une immatriculation au tableau d'immatriculations d'une voiture

```

1 void voi_ecrireFichier(const_Voiture self , FILE* fd)
2 {
3     //On écrit la valeur de l'année puis du nombre de km:
4     fwrite(&(self->year) , sizeof(int) , 1 , fd);
5     fwrite(&(self->km) , sizeof(int) , 1 , fd);
6
7     //On écrit la longueur de la chaîne de caractère de la marque
8     //puis la marque elle-même (grâce à l'information sur la longueur
9     //d'ailleurs!):
10    int brand_len = strlen(self->brand) + 1; //+1 pour le '\0'...
11    fwrite(&brand_len , sizeof(int) , 1 , fd);
12    fwrite(self->brand , sizeof(char) , brand_len , fd);
13
14    //On fait la même chose que pour la marque sauf qu'on écrit en
15    //plus la longueur du tableau de chaînes et on itère pour chaque
16    //élément du tableau qui est une chaîne:
17    fwrite(&(self->nbImmat) , sizeof(int) , 1 , fd);
18    int immat_len = 0;
19    for( int i = 0; i < self->nbImmat; i++)
20    {
21        immat_len = strlen(self->immats[i]) + 1; //+1 pour le '\0'...
22        fwrite(&immat_len , sizeof(int) , 1 , fd);
23        fwrite(self->immats[i] , sizeof(char) , immat_len , fd);
24    }
25 }
```

**Listing 13.26** – Écriture dans un fichier de tous les champs d'une voiture

```

1 void voi_lireFichier(Voiture self, FILE* fd)
2 {
3     //On lit dans l'ordre où on a écrit, donc d'abord l'année
4     //puis les km:
5     fread(&(self->year), sizeof(int), 1, fd);
6     fread(&(self->km), sizeof(int), 1, fd);
7
8
9     //On réinitialise self. Donc le tableau d'immatriculations de self
10    //peut contenir des valeurs qui ont été allouées dynamiquement (cas où
11    //self n'a pas été créé par voi_creerFrom Fichier par exemple).
12    //Comme on ne dispose pas de fonction voi_detruire() (on pourrait
13    //l'implanter) on doit vider ce qui doit être vidé dans voi_lireFichier
14    //():
15    if (self->brand != NULL)
16    {
17        free(self->brand);
18
19        for (int i = 0; i < self->nbImmat; i++)
20            free(self->immat[i]);
21
22
23    //Que self vienne de voi_creerFromFichier() ou d'une voiture qu'on
24    //réinitialise donc non vide on repasse nbImmat à 0 pour la fonction
25    //voi_addImmatriculation() en bas:
26    self->nbImmat = 0;
27
28
29    //On lit la marque, comme celle-ci peut avoir une longueur plus grande
30    //que la marque précédente on réalloue le champ à la bonne longueur:
31    int brand_len = 0;
32    fread(&brand_len, sizeof(int), 1, fd);
33    self->brand = (char*) malloc(brand_len*sizeof(char));
34    fread(self->brand, sizeof(char), brand_len, fd);
35
36
37    //On lit le tableau d'immatriculations qui est un tableau de chaînes
38    //de caractères. La variable n est la longueur de ce tableau
39    //d'immatriculations:
40    int immat_len = 0;
41    int n = 0;
42    fread(&(n), sizeof(int), 1, fd);
43    for (int i = 0; i < n; i++)
44    {
45        fread(&immat_len, sizeof(int), 1, fd);
46        char tmp[immat_len];
47        fread(tmp, sizeof(char), immat_len, fd);
48        voi_addImmatriculation(self, tmp);
49    }
50
51 }

```

**Listing 13.27** – Lecture depuis un fichier des champs d'une voiture précédemment sauvegardés, on réinitialise une voiture passée en paramètre

### **13.4.2 Intégration de Collection.c de l'exercice et Voiture.c de l'exercice 4a**

En suivant la méthode donnée par ce TP (exercice 1, puis 2, puis 3, puis 4.1...) on obtient l'affichage dont l'extrait a été fourni précédemment à l'exercice 2.

# Chapitre 14

## Tubes, Forks et Exec

### Résumé et cadre de travail

Dans ce TP il s'agit de manipuler la communication entre deux processus lourds via des tubes nommés ou anonymes.

Comme effet collatéral, les fonctions fork et exec sont également étudiées.

Directive importante : pour tout appel système, (ouverture de fichier, lancement d'un thread, ...) vous devez tester la valeur de retour pour détecter toute anomalie ; on ne présuppose jamais qu'un appel système fonctionne. Un assert suffira amplement.

Autre directive : nous n'utiliserons que les entrées-sorties de bas niveau (open, write, ...).

# Principe de fonctionnement des tubes

## Point de cours

Deux processus lourds ne partagent pas, par définition, le même espace mémoire (à l'inverse des processus légers nommés également *threads*).

Une solution est d'utiliser un tube. Un tube est unidirectionnel ; généralement un processus le manipule en écriture pour injecter des données à l'intérieur, et un autre le manipule en lecture et récupère les données injectées par le premier processus.

On parle d'écrivain et de lecteur, ou encore de producteur et de consommateur.

Dans le fonctionnement par défaut la lecture est bloquante : un processus effectuant un ordre de lecture sur un tube vide se bloque jusqu'à ce qu'une donnée soit injectée.

L'écriture dans un tube est légèrement différente. En effet un tube a une capacité limitée (64 Ko sur mon système) et un processus écrivain peut se retrouver devant deux situations :

- La capacité du tube n'est pas atteinte et l'écrivain dépose ses données et continue son code, même si aucun lecteur n'est présent.
- La capacité du tube est atteinte et l'écrivain se bloque jusqu'à ce que suffisamment de place se libère (due à un lecteur qui a consommé).

Ceci dit, il ne faut jamais que le bon fonctionnement d'un programme dépende de la capacité d'un tube. Une autre façon de présenter les choses est de s'assurer qu'un programme fonctionne dans le cas où la capacité est réduite à zéro ; autrement dit de presupposer qu'un processus ne peut écrire que lorsqu'un lecteur fait une demande de lecture.

Les tubes nommés apparaissent dans le système de fichiers et se manipulent, par les processus, comme des fichiers "normaux". Mais ce n'est qu'apparence : si le processus l'ouvre en écriture c'est un producteur, sinon c'est un consommateur. Et il n'est donc pas logique de l'ouvrir à la fois en lecture et en écriture <sup>a</sup>.

Les tubes anonymes sont créés par les processus et ont comme durées de vie celles des processus qui les manipulent. Ils n'apparaissent pas dans le système de fichiers et ne peuvent être manipulés que par des processus issus du même *fork* (cf. exercices dédiés).

---

a. Ceci dit, c'est techniquement possible sur mon système, ce qui n'est pas une raison pour le faire.

## Bash et tubes nommés

---

**Question :** La commande

```
$ mkfifo montube
```

crée un tube nommé montube. En listant les fichiers vérifiez les droits par défaut et que le type du fichier est particulier.

**Réponse :**

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX3$ mkfifo montube
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX3$ ls
| montube |
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX3$ ls -l
total 0
prw-rw-r-- 1 crex crex 0 nov. 14 13:56 |montube|
```

On constate que le type du fichier est 'p' pour 'pipe'. Il y a des droits en lecture/écriture pour le propriétaire du pipe, lecture/écriture pour les groupes et lecture seule pour tout autre utilisateur.

---

**Question :** Lancez 3 terminaux positionné dans le répertoire contenant le tube. Deux lisent dans le tube avec la commande

```
$ cat < montube
```

(ils sont donc en concurrence) et un écrit dans le tube avec la commande

```
$ cat > montube
```

Écrivez plusieurs lignes et regardez comment les lectures se répartissent. Quel que soit le résultat, vous ne pouvez faire aucune hypothèse sur cette répartition.

**Réponse :**

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX3$ ls
montube
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX3$ cat > montube
hello world
You?
how are you?
Currently testing tubes with one writer and two readers, who's gonna read first?
You?
Or you?
[]

[Output from the two readers]
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX3$ cat < montube
hello world
You?
how are you?
Currently testing tubes with one writer and two readers, who's gonna read first?
Or you?

[Output from the writer]
```

L'écrivain est dans la moitié gauche de l'écran, les deux lecteurs dans la moitié droite. Les deux lecteurs semblent se partager la lecture à tout de rôle mais il vaut mieux ne pas trop faire confiance en ce comportement (ils sont en concurrence...)

---

---

**Question :** Arrêtez l'écrivain (avec Ctrl-D ou Ctrl-C) et notez que les lecteurs se terminent également. Vérifiez que le tube existe toujours dans le système de fichiers.

**Réponse :** Après avoir arrêté l'écrivain avec Ctrl-C les deux lecteurs se sont également arrêtés (les lecteurs sont certes bloqués s'il n'y a pas d'écriture mais s'il n'y a pas d'écrivain il est inutile de les faire attendre) et le tube existe encore dans le système de fichiers après utilisation car il est nommé.

---

**Question :** Dans un terminal, envoyez la chaîne "bonjour" dans le tube avec la commande

```
$ echo bonjour > montube
```

On remarque que la commande est bloquée ce qui semble entrer en contradiction avec le fait qu'un tube a un tampon : nous verrons par la suite (en programmant) que c'est l'ouverture du tube qui bloque et non l'écriture. Lancez un lecteur avec

```
$ cat < montube
```

pour vérifier que tout se débloque.

**Réponse :**

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX3$ echo bonjour > montube
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX3$ echo bonjour > montube
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX3$ cat < montube
bonjour
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX3$
```

---

**Question :** Effacez le tube.

**Réponse :** Un simple "rm <nom\_du\_tube>" suffit comme pour un fichier...

---

## Programmation et tubes nommés

Rappelez-vous qu'il faut tester (avec un assert par exemple) systématiquement le retour de chaque appel système. Dans cet exercice le tube pré-existe dans le système de fichiers.

**Question :** Faites un programme qui :

- ouvre en écriture le tube nommé dont le nom est passé en paramètre.
- affiche le message "le tube vient d'être ouvert en écriture".
- en boucle lit un caractère au clavier et l'écrit dans le tube. On pourra stopper la boucle sur un caractère particulier (de votre choix).
- ferme le tube

Lancez le programme et vérifiez bien que l'ouverture est bloquante. Lancez dans un autre terminal un lecteur en mode bash

```
$ cat < montube
```

Pour tester votre programme.

**Réponse :**

The screenshot shows a terminal session. On the left, the source code of a C program is displayed. It includes headers for stdio.h, stdlib.h, assert.h, sys/types.h, sys/stat.h, and fcntl.h. The main function checks if there's one argument, opens a file descriptor for the second argument in write-only mode, and then enters a loop reading from standard input and writing to the file descriptor. If the character read is a tilde (~), it exits. Otherwise, it writes the character to the file descriptor. Finally, it closes the file descriptor. On the right, the terminal session shows the command mkfifo montube being run, followed by the execution of the program ./ecrivain. The program outputs "Le tube vient d'être ouvert en écriture!". Then, in another terminal window, the command cat < montube is run, and the word "Hello" is typed, which appears in the first terminal window, demonstrating that the program is writing to the named pipe.

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

//Pour la fonction open():
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

//Pour la fonction write() et close():
#include <unistd.h>

int main(int argc, char *argv[])
{
    if(argc != 2)
    {
        fprintf(stderr, "One parameter only: tube name");
        exit(EXIT_FAILURE);
    }

    int fd = open(argv[1], O_WRONLY);
    assert(fd != -1);
    printf("Le tube vient d'être ouvert en écriture!\n");

    char c = 'o';
    while(c != '~')
    {
        scanf("%c", &c);
        write(fd, &c, sizeof(char));
    }

    close(fd);
}
```

Lorsque j'ai lancé ./ecrivain montube, l'ouverture ne s'est pas faite tant qu'un lecteur (cf. la commande "cat < montube") n'était pas présente (elle était donc bloquante). Une fois le lecteur présent, le message "Le tube vient d'être ouvert en écriture" s'est affiché. J'ai pu alors écrire "Hello" qui s'est affiché dans le lecteur dessous. Le caractère '~', désigné comme caractère de terminaison de la lecture a permis la fermeture de l'écrivain comme anticipé.

**Note :** Depuis l'édition de cette illustration immédiatement ci-dessus la condition dans la boucle while a été modifiée comme suit afin de gérer le cas d'un Ctrl+D (signal EOF) au clavier :

```
1 //On compte sur l'évaluation paresseuse
2 //en cas de EOF et de '~' en premier caractère:
3 while(scanf("%c", &c) != EOF && c != '~')
4 {
5     write_ret = write(fd, &c, sizeof(char));
6     assert(write_ret != -1);
7 }
```

---

**Question :** De même faites un programmeur qui lit dans le tube caractère par caractère et les affiche à l'écran ; testez-le avec un écrivain en mode bash

```
$ cat > montube
```

Comment réagit le lecteur si on arrête l'écrivain ? Détectez et gérez ce cas de figure dans votre programme.

**Réponse :**

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

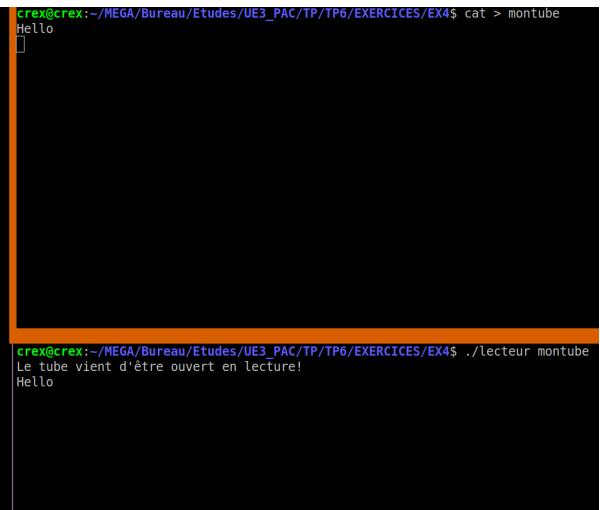
//Pour la fonction open():
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

//Pour la fonction write() et close():
#include <unistd.h>

int main(int argc, char *argv[])
{
    if(argc != 2)
    {
        fprintf(stderr, "One parameter only: tube name");
        exit(EXIT_FAILURE);
    }

    int fd = open(argv[1], O_RDONLY);
    assert(fd != -1);
    printf("Le tube vient d'être ouvert en lecture!\n");

    char c = 'o';
    while(c != '-')
    {
        read(fd, &c, sizeof(char));
        fprintf(stdout, "%c", c);
    }
    close(fd);
}
```



```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ cat > montube
Hello
^C
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ ./lecteur montube
Le tube vient d'être ouvert en lecture!
Hello
```

Si on lance la commande du lecteur avant l'écrivain, le tube n'est pas ouvert tant qu'on ne lance pas la commande l'écrivain du dessus (cat > montube). Une fois l'écrivain lancé, on peut commencer l'écriture et le lecteur affiche bien ce qui est écrit. Le caractère '~' arrête la lecture et ferme le lecteur.

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

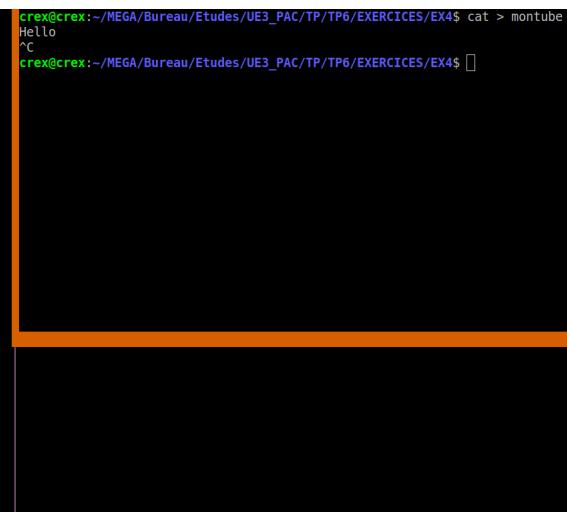
//Pour la fonction open():
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

//Pour la fonction write() et close():
#include <unistd.h>

int main(int argc, char *argv[])
{
    if(argc != 2)
    {
        fprintf(stderr, "One parameter only: tube name");
        exit(EXIT_FAILURE);
    }

    int fd = open(argv[1], O_RDONLY);
    assert(fd != -1);
    printf("Le tube vient d'être ouvert en lecture!\n");

    char c = 'o';
    while(c != '-')
    {
        read(fd, &c, sizeof(char));
        fprintf(stdout, "%c", c);
    }
    close(fd);
}
```



```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ cat > montube
Hello
^C
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$
```

Le problème survient lorsqu'on arrête l'écrivain avant le lecteur... Ce-dernier reste ouvert ! Mais on ne peut plus le fermer proprement avec '~'... Pour corriger ce problème on modifie le code visible sur la moitié gauche de l'illustration précédente comme ci-dessous :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4
5 //Pour la fonction open():
6 #include <sys/types.h>
7 #include <sys/stat.h>
8 #include <fcntl.h>
9
10 //Pour la fonction write() et close():
11 #include <unistd.h>
12
13 int main(int argc, char *argv[])
14 {
15     if(argc != 2)
16     {
17         fprintf(stderr, "One parameter only: tube name");
18         exit(EXIT_FAILURE);
19     }
20
21     int fd = open(argv[1], O_RDONLY);
22     assert(fd != -1);
23     printf("Le tube vient d'être ouvert en lecture!\n");
24
25     char c;
26     int read_return = 1;
27
28     while(c != '~' && read_return != 0)
29     {
30         read_return = read(fd, &c, sizeof(char));
31         assert(read_return != -1);
32         fprintf(stdout, "%c", c);
33     }
34
35     close(fd);
36 }
```

**Listing 14.1** – code source de lecteur.c après correction

En effet, dans le manuel Bash dédié à read (cf. man 3 read), on peut lire ceci : "If no process has the pipe open for writing, read() shall return 0 to indicate end-of-file." En d'autres termes, si la valeur de retour de la fonction read() vaut 0 alors c'est qu'il n'y a plus d'écrivain dans le tube. Après expérimentation, le lecteur s'interrompt effectivement correctement dès qu'on arrête brutalement l'écrivain.

---

**Question :** Testez vos deux programmes ensemble.

**Réponse :**

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ ./ecrivain montube
Le tube vient d'être ouvert en écriture!
Hello!
How are you?
~
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ ./ecrivain montube
Le tube vient d'être ouvert en écriture!
^C
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ ./ecrivain montube
Le tube vient d'être ouvert en écriture!
```

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ ./lecteur montube
Le tube vient d'être ouvert en lecture!
Hello!
How are you?
-
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ ./lecteur montube
Le tube vient d'être ouvert en lecture!
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ ./lecteur montube
Le tube vient d'être ouvert en lecture!
^C
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ 
```

On constate sur la dernière manipulation de l'illustration dessus que l'écrivain reste ouvert si le lecteur est brutalement arrêté. Cependant, la moindre écriture suffit à l'arrêter. La valeur de ce qui a été écrit n'est pas mémorisé dans le tube. Il semble donc qu'en l'absence de lecteurs l'écriture échoue et cet échec provoque la mort de l'écrivain.

---

**Question :** Lancez vos deux programme et commencez à les utiliser. Puis dans une troisième console, supprimez physiquement le tube. Et continuez à utiliser les deux programmes. Que se passe-t-il ?

**Réponse :**

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ ./ecrivain montube
Le tube vient d'être ouvert en écriture!
Hello
How are you?
]

[
```

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ ./lecteur montube
Le tube vient d'être ouvert en lecture!
Hello
How are you?

[
```

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ rm montube
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX4$ 
```

Dans l'illustration ci-dessus on a lancé l'écrivain et le lecteur. On a écrit "Hello" avec l'écrivain et le lecteur l'a lu puis affiché. On a ensuite supprimé le tube (cf. commande "rm montube" en bas à droite.) On a alors écrit "How are you ?" dans l'écrivain et... Le lecteur a lu et affiché "How are you ?" en suivant. Il semblerait donc que même en cas de suppression du tube, ce-dernier subsiste dans le système tant que celui-ci est encore référencé/utilisé par des processus.

---

# Programmation et tubes anonymes

## Point de cours

Un tube anonyme a le même fonctionnement qu'un tube nommé, mais :

- il n'apparaît pas sur le système de fichiers,
- il a une durée de vie liée à celle des programmes le manipulant.

En règle général, le déroulement est le suivant :

- le programme crée un tube anonyme (fonction système *pipe*)
- il se duplique (fonction système *fork*) et le tube est alors présent dans chaque processus
- un processus ferme l'extrémité du tube en lecture, et l'autre ferme celle en écriture
- alors l'un écrit et l'autre lit
- à la fin, chaque processus ferme l'extrémité qu'il avait conservée.

Dans cet exercice vous êtes en mode “recherche autonome” pour construire votre programme :

- rappelez-vous comment marchent les fonctions fork et wait,
- et étudiez la fonction pipe.

---

**Question :** Écrivez un tel programme. C'est le père qui lit dans le tube, et le fils qui écrit. Par exemple le fils envoie la chaîne “bonjour” et le père la récupère et l'affiche. Au fait comment le père connaît la longueur de la chaîne qu'il reçoit ?

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4 #include <string.h>
5
6 //Pour la fonction open():
7 #include <sys/types.h> //fork() a aussi besoin de ça
8 #include <sys/stat.h>
9 #include <fcntl.h>
10
11 #include <sys/wait.h>
12
13 //Pour la fonction write(), read(), close()
14 //et fork():
15 #include <unistd.h>
16
17 void pere(int fds[]);
18 void fils(int fds[]);
19
20 int main()
21 {
22     int fds[2];
23     int pipe_ret = pipe(fds);
24     assert(pipe_ret != -1);
25
26     if(fork() == 0)
27         fils(fds);
```

```

28
29     else
30     {
31         pere( fds );
32         wait( NULL );
33     }
34
35     return EXIT_SUCCESS;
36 }
37
38 void pere( int fds [] )
39 {
40     close( fds [ 1 ] );
41
42     int str_len;
43     int read_ret = read( fds [ 0 ] , &str_len , sizeof( int ) );
44     assert( read_ret != -1 );
45
46     char str[ str_len ];
47     for( int i = 0; i < str_len; i++ )
48     {
49         read_ret = read( fds [ 0 ] , &(str[ i ]) , sizeof( char ) );
50         assert( read_ret != -1 );
51     }
52
53     fprintf( stdout , "%s\n" , str );
54
55     close( fds [ 0 ] );
56 }
57
58 void fils( int fds [] )
59 {
60     close( fds [ 0 ] );
61
62     char str[] = "Bonjour!";
63     int str_len = strlen( str ) + 1; //+1 pour le '\0'
64
65     int write_ret = write( fds [ 1 ] , &str_len , sizeof( int ) );
66     assert( write_ret != -1 );
67
68     for( int i = 0; i < str_len; i++ )
69     {
70         write_ret = write( fds [ 1 ] , &(str[ i ]) , sizeof( char ) );
71         assert( write_ret != -1 );
72     }
73
74     close( fds [ 1 ] );
75 }
```

**Listing 14.2** – Code source de tube.c

Pour que le père, lecteur, connaisse la taille de la chaîne à lire, on a fait le choix de demander au fils d'envoyer la taille de cette chaîne au père dans le tube.

## Programmation et tubes anonymes : bidirectionnel

**Question :** Si l'on veut avoir une communication bidirectionnelle, il est nécessaire d'avoir deux tubes. Écrivez un programme qui :

- crée les deux tubes
- se duplique.
- le père lit deux entier au clavier et les envoie au fils
- le fils récupère les deux entiers et envoie la somme au père qui l'affiche
- le programme doit se terminer proprement, notamment en ce qui concerne la fermeture des descripteurs de fichiers.

Note : on pourrait faire la même chose avec deux tubes nommés.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4 #include <string.h>
5
6 //Pour la fonction open():
7 #include <sys/types.h> //fork() a aussi besoin de ça
8 #include <sys/stat.h>
9 #include <fcntl.h>
10
11 #include <sys/wait.h>
12
13 //Pour la fonction write(), read(), close()
14 //et fork():
15 #include <unistd.h>
16
17 void pere(int fds1[], int fds2[]);
18 void fils(int fds1[], int fds2[]);
19
20 int main()
21 {
22     int fds1[2];
23     int pipe_ret1 = pipe(fds1);
24     assert(pipe_ret1 != -1);
25
26     int fds2[2];
27     int pipe_ret2 = pipe(fds2);
28     assert(pipe_ret2 != -1);
29
30
31     if(fork() == 0)
32         fils(fds1, fds2);
33
34     else
35     {
36         pere(fds1, fds2);
37         wait(NULL);
38     }
39
40     return EXIT_SUCCESS;
41 }
```

```

42 void pere( int fds1[], int fds2[])
43 {
44     //Le père est lecteur sur fds1, il reçoit de fds1[0]:
45     close(fds1[1]);
46
47     //Il est écrivain sur fds2, il envoie sur fds2[1]:
48     close(fds2[0]);
49
50
51
52     //Le père lit deux entiers au clavier et les envoie au fils:
53     int a, b;
54     scanf( "%i %i" , &a , &b );
55
56     write( fds2[1] , &a , sizeof(int));
57     write( fds2[1] , &b , sizeof(int));
58
59
60     //Le père reçoit la somme du fils et affiche le résultat:
61     int c;
62     read( fds1[0] , &c , sizeof(int));
63     fprintf(stdout , "Le résultat est: %i\n" , c );
64
65
66     close(fds1[0]);
67     close(fds2[1]);
68 }
69
70 void fils( int fds1[], int fds2[])
71 {
72     //Le fils est écrivain sur fds1, il envoie sur fds1[1]:
73     close(fds1[0]);
74
75     //Le fils est lecteur sur fds2, il reçoit sur fds2[0]:
76     close(fds2[1]);
77
78
79     //Le fils récupère les deux entiers:
80     int a, b;
81     read( fds2[0] , &a , sizeof(int));
82     read( fds2[0] , &b , sizeof(int));
83     fprintf(stdout , "On fait l'addition de %i et %i...\\n" , a , b );
84
85
86     //Le fils envoie la somme au père:
87     int c = a + b;
88     write( fds1[1] , &c , sizeof(int));
89
90
91     close(fds1[1]);
92     close(fds2[0]);
93 }
```

**Listing 14.3** – Code source de bitube.c

On notera qu'on devrait tester à chaque `read()` et `write()` que tout s'est bien passé avec un `assert(...)`. Cela nuirait à la lisibilité mais nous pourrions créer des fonctions annexes pour éviter cela.

## Exec

### Point de cours

La famille des fonctions *exec* permet de substituer le code du processus courant par le code d'un autre exécutable. L'exécutable est désigné par son nom de fichier.

Il est possible de passer des paramètres au nouveau programme comme si on les avait tapés en ligne de commande.

Les fonctions de type *exec* sont souvent associées à un *fork*.

Note : nous utiliserons uniquement la fonction *execv* dans la suite du TP.

---

### Question :

Écrivez un programme qui :

- prend plusieurs paramètres en ligne de commande
- se substitue à la commande /bin/ls en lui passant les mêmes paramètres.

### Réponse :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 //Pour exec():
5 #include <unistd.h>
6
7 int main(int argc, char* argv[])
8 {
9     if(argc <= 1)
10    {
11        fprintf(stderr, "Not enough arguments...\\n");
12        exit(EXIT_FAILURE);
13    }
14
15    execv("/bin/ls", argv);
16
17    return EXIT_SUCCESS;
18 }
```

**Listing 14.4** – Code source de myls.c

---

---

**Question :**

Écrivez un programme qui affiche la liste de ses arguments. Puis reprenez le premier programme de l'exercice qui au lieu de se substituer avec la commande /bin/ls se substitue avec ce nouveau programme.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main( int argc , char *argv [] )
5 {
6     for( int i = 0; i < argc; i++)
7         fprintf(stdout , "arg %i is: %s\n" , i , argv[ i ] );
8
9     return EXIT_SUCCESS;
10 }
```

**Listing 14.5** – Code source du programme affichant ses arguments, printArgs.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 //Pour exec():
5 #include <unistd.h>
6
7 int main( int argc , char* argv [] )
8 {
9     if( argc <= 1)
10     {
11         fprintf(stderr , "Not enough arguments...\n");
12         exit(EXIT_FAILURE);
13     }
14
15     execv( "/home/crex/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX7/
16     printArgs" ,
17             argv );
18
19     return EXIT_SUCCESS;
}
```

**Listing 14.6** – Code source du programme affichant ses arguments en utilisant printArgs, myArgsPrinter.c

---

---

**Question :**

Écrivez un programme qui prend deux arguments qui sont des entiers et en affiche la somme.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[])
5 {
6     if (argc != 3)
7     {
8         fprintf(stderr, "An addition requires two and only two operands...\n");
9         exit(EXIT_FAILURE);
10    }
11
12    //Converting char to int from argv array and computing sum:
13    int a = strtol(argv[1], NULL, 10);
14    int b = strtol(argv[2], NULL, 10);
15    int c = a + b;
16
17    //Printing the result:
18    fprintf(stdout, "%i + %i = %i\n", a, b, c);
19
20    return EXIT_SUCCESS;
21 }
```

**Listing 14.7** – Code source du programme faisant la somme de deux nombres passés en argument, adder.c

---

**Question :**

Écrivez un programme qui :

- lit deux entiers au clavier (avec scanf) ; il s'agit bien de stocker ces deux entiers dans des variables de type int.
- se substitue au programme précédent en lui passant les deux entiers en paramètres.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/wait.h>
4 #include <assert.h>
5
6 //Pour exec():
7 #include <unistd.h>
8
9 #define NB_MAXLEN 10
10
11 int main()
12 {
13     char a[NB_MAXLEN];
```

```
14 |     char b[NB_MAXLEN];
15 |     scanf( "%s %s ", a, b );
16 |
17 |     char* path = "/home/crex/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/
18 | EX7/adder";
19 |     char* operands[] = {path, a, b, NULL};
20 |
21 |     execv(path, operands);
22 |
23 |     return EXIT_SUCCESS;
24 }
```

**Listing 14.8** – Code source du programme faisant la somme de deux nombres passés entrés au clavier en utilisant le programme adder précédent

**Question :** Que se passe-t-il si un programme fait un exec sur lui-même ?

**Réponse :** Il remplace son code source au moment de l'appel de exec par son code source et rentre ainsi dans une boucle récursive sans terminaison. En témoigne l'illustration d'une expérimentation ci-dessous :

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

//Pour exec():
#include <unistd.h>

int main(int argc, char* argv[])
{
    printf("%i\n", argc);
    execv("/home/crex/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/EX7/test", argv);

    return EXIT_SUCCESS;
}
```

On remarque ainsi que le "return EXIT\_SUCCESS" n'est jamais atteint et qu'on doit interrompre le signal par un signal avec "Ctrl+C".

## Fork, exec et tubes anonymes

### Point de cours

À la suite d'un *exec*, tous les file descriptors préalablement ouverts (notamment les *pipes*) restent ouverts dans le nouveau code.

Le nouveau code n'en a priori pas conscience : mais il est programmé en ce sens et donc ce n'est pas un problème. Le vrai problème est qu'il ignore leurs valeurs (on rappelle qu'un file descriptor de bas niveau est de type *int*).

Il faut donc, d'une manière ou d'une autre, indiquer au nouveau code les valeurs de ces file descriptors ; par exemple en arguments de la ligne de commandes.

### Question :

Écrivez un programme qui :

- prend un paramètre qui est la valeur du file descriptor dans lequel il va écrire
- lit deux entiers en clavier
- les écrit dans le file descriptor en mode binaire
- ferme le file descriptor,
- se termine

### Réponse :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4
5 //Pour write():
6 #include <unistd.h>
7
8 int main(int argc, char* argv[])
9 {
10     if(argc != 2)
11     {
12         fprintf(stderr, "Program parameter must be one file descriptor\n");
13         exit(EXIT_FAILURE);
14     }
15
16     int fd = (int) strtol(argv[1], NULL, 10);
17     printf("Writer fd is %i\n", fd);
18
19     int a, b;
20     scanf("%i %i", &a, &b);
21
22     int write_ret;
23     write_ret = write(fd, &a, sizeof(int));
24     assert(write_ret != -1);
25     write_ret = write(fd, &b, sizeof(int));
26     assert(write_ret != -1);
27
28     close(fd);
29 }
```

```
30     return EXIT_SUCCESS;
31 }
```

**Listing 14.9** – Code source du programme écrivant deux nombres saisis par l'utilisateur au clavier dans un file descriptor passé en paramètre

---

**Question :**

Écrivez un programme qui :

- prend un paramètre qui est la valeur du file descriptor dans lequel il va lire
- lit deux entiers dans le file descriptor en mode binaire
- ferme le file descriptor,
- affiche la somme des deux entiers
- se termine

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4
5 //Pour read():
6 #include <unistd.h>
7
8 int main(int argc, char* argv[])
9 {
10     if(argc != 2)
11     {
12         fprintf(stderr, "Program parameter must be one file descriptor\n");
13         exit(EXIT_FAILURE);
14     }
15
16     int fd = (int) strtol(argv[1], NULL, 10);
17     printf("Reader fd is %i\n", fd);
18
19     int a, b;
20     int read_ret;
21     read_ret = read(fd, &a, sizeof(int));
22     assert(read_ret != -1);
23     read_ret = read(fd, &b, sizeof(int));
24     assert(read_ret != -1);
25
26     close(fd);
27
28     fprintf(stdout, "%i + %i = %i\n", a, b, a + b);
29
30     return EXIT_SUCCESS;
31 }
```

**Listing 14.10** – Code source du programme lisant deux nombres dans un file descriptor passé en paramètre

---

---

**Question :**

Écrivez un troisième programme qui :

- crée un tube anonyme
- se duplique avec un fork
- le fils ferme l'extrémité en écriture et remplace son code par le second programme en lui passant la valeur du file descriptor encore ouvert
- le père ferme l'extrémité en lecture et remplace son code par le premier programme en lui passant la valeur du file descriptor encore ouvert

Vérifier que tout se passe bien.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4 #include <string.h>
5
6 //Pour la fonction open():
7 #include <sys/types.h> //fork() a aussi besoin de ça
8 #include <sys/stat.h>
9 #include <fcntl.h>
10
11 #include <sys/wait.h>
12
13 //Pour la fonction write(), read(), close()
14 //et fork():
15 #include <unistd.h>
16
17 void pere(int fds[]);
18 void fils(int fds[]);
19
20 int main()
21 {
22     int fds[2];
23     int pipe_ret = pipe(fds);
24     assert(pipe_ret != -1);
25
26     if(fork() == 0)
27         fils(fds);
28
29     else
30         pere(fds);
31
32     return EXIT_SUCCESS;
33 }
34
35 void pere(int fds[])
36 {
37     close(fds[0]);
38
39     char* path = "/home/crex/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/
40     EX8/fdWriter";
41     char fd[50];
```

```

41     sprintf(fd , "%i" , fds [1]) ;
42     char* argv [] = {path , fd , NULL} ;
43
44     execv(path , argv) ;
45 }
46
47 void fils(int fds [])
48 {
49     close(fds [1]) ;
50
51     char* path = "/home/crex/MEGA/Bureau/Etudes/UE3_PAC/TP/TP6/EXERCICES/
52     EX8/fdReader" ;
53     char fd[50] ;
54     sprintf(fd , "%i" , fds [0]) ;
55     char* argv [] = {path , fd , NULL} ;
56
57     execv(path , argv) ;

```

**Listing 14.11** – Code source du programme réalisant l’addition complète en orchestrant la communication entre le père et le fils

**Question :** dans le troisième programme, pourquoi le père, après le execv, ne peut-il pas faire un wait ?

**Réponse :** Le execv remplace le code du père par le code correspondant au programme donné en paramètre de execv. Une instruction wait() serait donc écrasée ! De plus, ce programme (ici fdWriter) ne contient pas l'instruction wait(). **Pourrait-on cependant le faire ?** Le père ne peut donc pas faire un wait(). **Cependant, nous savons que la création d'un tube implique une écriture/lecture bloquante. Tant qu'il n'y a pas de lecteur l'écrivain attend et tant qu'il n'y a pas d'écrivain, le lecteur attend.** Ce n'est donc pas un problème !

## Fork, exec et tubes nommés

Le but est de reprendre tout ce qui a été vu jusqu'à présent. On demande à ce que tous les appels systèmes soit testés (un assert est pleinement suffisant).

Il y a plusieurs paires worker/master, chaque programme étant indépendant c'est à dire qu'il possède son propre code source et son propre exécutable.

Un master reçoit deux arguments en lignes de commandes :

- le nom du tube lui permettant d'envoyer des données à son worker,
- le nom du tube lui permettant de recevoir le ou les résultats du worker.

Un worker a exactement les mêmes paramètres. Il y a un protocole de communication qui est propre à chaque paire en fonction des calculs demandés. Il y a un programme principal (l'orchestre) dont le but est de lancer les workers et les masters.

### 14.9.1 Chef d'orchestre

Le fonctionnement du chef d'orchestre est le suivant :

- il demande à l'utilisateur un numéro compris entre 1 et le nombre de paires master/worker,
- crée les deux tubes nommés,
- lance les master et worker désignés qui communiqueront avec les deux tubes
- attend la fin des deux programmes
- efface les tubes
- recommence

Pour simplifier, on suppose que les noms des masters et des workers sont codifiés :

- masters : "master <n>" où n varie de 1 au nombre de masters,
- "worker <n>" où n varie de 1 au nombre de workers,

Au lancement du chef d'orchestre, en ligne de commandes, on lui indique le nombre de paires master/worker. Par convention, le numéro 0 signifie que l'on arrête l'orchestre. On demande à ce que les tubes aient des noms uniques et clairs. On entend par "unique" que le même nom ne doit pas être utilisé deux fois. Supposons par exemple qu'on lance 3 paires de worker/master avec les numéros 4, 15, 8 ; les noms pourraient être :

- pipeMasterToWorker\_1\_4 et pipeWorkerToMaster\_1\_4
- pipeMasterToWorker\_2\_15 et pipeWorkerToMaster\_2\_15
- pipeMasterToWorker\_3\_8 et pipeWorkerToMaster\_3\_8

Note : comme deux paires master/worker ne tournent pas en même temps, on aurait pu réutiliser les mêmes noms ; la contrainte est purement pédagogique.

Réponse :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4 #include <errno.h>
5 #include <string.h>
6
7 //Pour la fonction open():
8 #include <sys/types.h> //fork() a aussi besoin de ça
9 #include <sys/stat.h>
10 #include <fcntl.h>
11
12 #include <sys/wait.h>
```

```

13 //Pour la fonction write() , read() , close()
14 //et fork():
15 #include <unistd.h>
16
17 void pere(const char writeTube[] , const char readTube[]);
18 void fils(const char writeTube[] , const char readTube[]);
19
20 int main()
21 {
22     int nb_pairs = 0;
23
24     do{
25         //Getting nb of pairs of master/worker...
26         //If 0 do...while() loop will stop:
27         printf("Enter a number of pairs of master/worker: ");
28         scanf("%i" , &nb_pairs);
29         printf("\n");
30
31         for(int i = 0; i < nb_pairs; i++)
32         {
33             int mkfifo_ret;
34             mkfifo_ret = mkfifo("pipeMasterToWorker" , 0600);
35             assert(mkfifo_ret != -1);
36             mkfifo_ret = mkfifo("pipeWorkerToMaster" , 0600);
37             assert(mkfifo_ret != -1);
38             printf("Pipes created successfully !\n");
39
40             int fork_ret = fork();
41             assert(fork_ret != -1);
42             if(fork_ret == 0)
43             {
44                 fils("pipeWorkerToMaster" , "pipeMasterToWorker");
45             }
46
47             fork_ret = fork();
48             assert(fork_ret != -1);
49             if(fork_ret == 0)
50             {
51                 pere("pipeMasterToWorker" , "pipeWorkerToMaster");
52             }
53
54             wait(NULL);
55             wait(NULL);
56             unlink("pipeMasterToWorker");
57             unlink("pipeWorkerToMaster");
58         }
59     }
60     while(nb_pairs > 0);
61
62     return EXIT_SUCCESS;
63 }
64
65 void pere(const char writeTube[] , const char readTube[])
66 {
67     int fdw = open(writeTube , O_WRONLY);
68

```

```

69 printf("Pere: %s\n", strerror(errno));
70 assert(fdw != -1);
71 int fdr = open(readTube, O_RDONLY);
72 assert(fdr != -1);
73
74 char fdW[50];
75 sprintf(fdW, "%i", fdw);
76 char fdR[50];
77 sprintf(fdR, "%i", fdr);
78
79 char* path = "./adderM";
80 char* argv[] = {path, fdW, fdR, NULL};
81
82 execv(path, argv);
83 assert(false);
84 }
85
86 void fils(const char writeTube[], const char readTube[])
87 {
88 int fdr = open(readTube, O_RDONLY);
89 printf("Fils: %s\n", strerror(errno));
90 assert(fdr != -1);
91 int fdw = open(writeTube, O_WRONLY);
92 assert(fdw != -1);
93
94 char fdW[50];
95 sprintf(fdW, "%i", fdw);
96 char fdR[50];
97 sprintf(fdR, "%i", fdr);
98
99 char* path = "./adderW";
100 char* argv[] = {path, fdW, fdR, NULL};
101
102 execv(path, argv);
103 assert(false);
104 }
```

**Listing 14.12** – Code source du chef d'orchestre

### 14.9.2 Worker/master 1

Le master demande deux entiers à l'utilisateur, les envoie au worker qui retourne la somme.

Réponse :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <assert.h>
5
6 #include <sys/types.h>
7 #include <sys/stat.h>
8 #include <fcntl.h>
9
10 int main(int argc, char* argv[])
11 {
12     if(argc != 3)
13     {
14         fprintf(stderr, "Error, wrong number of arguments: two tube names
15 required\n");
16         exit(EXIT_FAILURE);
17     }
18     int fdw = (int) strtol(argv[1], NULL, 10);
19     int fdr = (int) strtol(argv[2], NULL, 10);
20
21     int a, b;
22     printf("Enter two numbers: ");
23     scanf("%i %i", &a, &b);
24     printf("\n");
25
26     int write_ret;
27     write_ret = write(fdw, &a, sizeof(int));
28     assert(write_ret != -1);
29     write_ret = write(fdw, &b, sizeof(int));
30     assert(write_ret != -1);
31
32     int read_ret;
33     int c;
34     read_ret = read(fdr, &c, sizeof(int));
35     assert(read_ret != -1);
36
37     printf("%i + %i = %i\n", a, b, c);
38
39     close(fdw);
40     close(fdr);
41
42     return EXIT_SUCCESS;
43 }
```

**Listing 14.13** – Code source du maître demandant deux nombres à l'utilisateur et envoyant ces deux nombres au fils pour qu'il fasse l'addition. Le maître retourne ensuite le résultat

Réponse :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <assert.h>
5
6 #include <sys/types.h>
7 #include <sys/stat.h>
8 #include <fcntl.h>
9
10 int main(int argc, char* argv[])
11 {
12     if(argc != 3)
13     {
14         fprintf(stderr, "Error, wrong number of arguments: two tube names
required\n");
15         exit(EXIT_FAILURE);
16     }
17
18     int fdw = (int) strtol(argv[1], NULL, 10);
19     int fdr = (int) strtol(argv[2], NULL, 10);
20
21     int read_ret;
22     int a, b;
23
24     read_ret = read(fdr, &a, sizeof(int));
25     assert(read_ret != -1);
26     read_ret = read(fdr, &b, sizeof(int));
27     assert(read_ret != -1);
28     printf("Worker reads: %d and %d\n", a, b);
29
30     int write_ret;
31     int c = a + b;
32
33     printf("Worker returns result %d to Master\n", c);
34     write_ret = write(fdw, &c, sizeof(int));
35     assert(write_ret != -1);
36
37     close(fdw);
38     close(fdr);
39
40     return EXIT_SUCCESS;
41 }
```

**Listing 14.14** – Code source de l'esclave qui reçoit deux nombres du maître, fait l'addition et renvoie le résultat au maître

### **14.9.3 Worker/master 2**

Le master demande une chaîne de caractères à l'utilisateur, l'envoie au worker qui retourne la même chaîne privée des voyelles. Les données transmises doivent être réduites au minimum et on ne fait aucune hypothèse sur la longueur des chaînes transmises.

#### **14.9.4 Worker/master 3**

À votre initiative !

# Chapitre 15

## Sémaphores

### Résumé et Cadre de travail

Dans ce TP il s'agit de manipuler la communication entre plusieurs processus lourds (ou non d'ailleurs) via des sémaphores IPC.

Il y a également une initiation aux segments de mémoire partagée et aux files de messages.

Rappel d'une directive importante : pour tout appel système, (ouverture de fichier, lancement d'un thread, ...) vous devez tester la valeur de retour pour détecter toute anomalie ; on ne présuppose jamais qu'un appel système fonctionne. Un assert suffira amplement.

Le fichier exécutable rmsem.sh détruit tous les sémaphores qui ont les droits 0641 (cf. section 3).

### Principe de fonctionnement des sémaphores IPC

#### Point de cours

Un sémaphore permet à plusieurs processus de se synchroniser. On note deux grands types de synchronisation :

- précédence : le code d'un processus doit impérativement s'exécuter avant le code d'un second
- barrière : en un point précis du code, les processus doivent s'attendre mutuellement avant de continuer leurs exécutions

Nous utiliserons obligatoirement les fonctions suivantes :

- *semget* : pour créer ou ouvrir un sémaphore
- *semop* : pour faire les actions courantes sur un sémaphore
- *semctl* : pour effectuer des opérations de "maintenance" (initialisation, destruction, ...)
- *ftok* : pour de générer facilement des clés uniques pour l'identification des sémaphores (ou autres objets IPC d'ailleurs).

Précision : *semget* crée en réalité un tableau de sémaphores, tableau dont la taille peut tout à fait être réduite à une case.

Pensez que les sémaphores IPC ne sont pas automatiquement détruits à la fin de l'exécution des processus. C'est donc bien aux processus de détruire explicitement les sémaphores inutiles. Si ce n'est pas fait, il faut effacer les sémaphores à la main dans la console de commandes (cf. commandes "*ipcs -s*", "*ipcrm -s*" et "*ipcrm -S*").

## Création et destruction de sémaphores

---

**Question :** La commande

```
$ ipcs -s
```

permet d'avoir la liste des sémaphores existants. Notez les bien par lorsqu'on en effacera à la main (i.e. dans une console) il ne faudra pas toucher à cette liste initiale (ce qui autrement risquerait d'altérer le fonctionnement d'autres processus).

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP7/CODE_ETU/EXO_03$ ipcs -s
----- Semaphore Arrays -----
key      semid      owner      perms      nsems
```

---

**Question :** Créez un programme C (squelette exo3b.c fourni) qui crée un tableau de deux sémaphores. La clé sera IPC\_PRIVATE et les droits seront : - rw- pour l'utilisateur - r- pour le groupe - -x pour les autres Ces droits farfelus (0641 donc) permettront d'identifier facilement vos sémaphores avec la commande "ipcs -s", et de les effacer automatiquement avec l'exécutable rmsem.sh.

Il est conseillé d'utiliser ces droits pour tout le reste du TP. Ne détruisez pas le sémaphore dans votre programme et vérifiez qu'il est bien présent après l'exécution de ce dernier.

**Réponse :**

```
1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdlib.h>
5 // mettre les bons includes
6
7 #include <sys/types.h>
8 #include <sys/IPC.h>
9 #include <sys/sem.h>
10 #include <assert.h>
11
12 int main()
13 {
14     // ici il faut déclarer un sémaphore (en mode IPC_PRIVATE) de deux
15     // cases
16     // et donc on ne le détruit pas
17     int semid = semget(IPC_PRIVATE, 2, 0641);
18     assert(semid != -1);
19
20     return EXIT_SUCCESS;
}
```

```
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP7/CODE_ETU/EXO_03$ ls
exo3b exo3b.c exo3d.c
(base) florian@CReXJr:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP7/CODE_ETU/EXO_03$ ipcs -s
----- Semaphore Arrays -----
key      semid      owner      perms      nsems
0x00000000 0          florian    641          2
```

**Question :** Enfin supprimez le sémaphore avec la commande “ipcrm -s”.

**Réponse :** Le semid de notre sémaphore (cf. illustration précédente) était 0. On entre donc la commande "ipcrm -s 0" et le sémaphore n'apparaît plus dans le tableau.

---

**Question :** Cette fois-ci le programme détruit le (tableau de) sémaphore(s). Le squelette exo3d.c est fourni. Après l'exécution, vérifiez (avec “ipcs -s”) que tout est correct

**Réponse :**

```
1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdlib.h>
5 // mettre les bons includes
6 #include <assert.h>
7 #include <sys/types.h>
8 #include <sys/ipc.h>
9 #include <sys/sem.h>
10
11 int main()
12 {
13     // ici il faut déclarer un sémaphore (en mode IPC_PRIVATE) de deux
14     // cases
15     // et on le détruit cette fois-ci
16     int semid = semget(IPC_PRIVATE, 1, 0641);
17     assert(semid != -1);
18
19     int semctl_ret = semctl(semid, 1, IPC_RMID);
20     assert(semctl_ret != -1);
21
22     return EXIT_SUCCESS;
}
```

---

## Utilisation basique d'un sémaphore

**Question :** Le squelette exo4.c vous est fourni. Le processus se duplique (fork) et chacun des deux processus fait un affichage. Mais il faut que celui du père se fasse avant celui du fils. Il faut donc implanter une précédence pour assurer ce fonctionnement. Le code fourni est commenté et vous indique quoi faire et à quels endroits.

Réponse :

```
1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <sys/types.h>
8 #include <sys/wait.h>
9 #include <sys/ipc.h>
10 #include <sys/sem.h>
11 #include <unistd.h>
12
13
14 void un(int semid)
15 {
16     sleep(3);
17     printf("Processus un %d ! (normalement je m'affiche en premier)\n",
18            getpid());
19
20     // ici il faut débloquer le processus "deux"
21     struct sembuf smb = {0, 1, 0};
22     int semop_ret = semop(semid, &smb, 1);
23     assert(semop_ret != -1);
24 }
25
26 void deux(int semid)
27 {
28     // ici il faut attendre le déblocage venant de "un"
29     struct sembuf smb = {0, -1, 0};
30     int semop_ret = semop(semid, &smb, 1);
31     assert(semop_ret != -1);
32
33     printf("Processus deux %d ! (normalement je m'affiche en dernier)\n",
34            getpid());
35 }
36
37 int main()
38 {
39     // ici il faut déclarer un sémaphore (en mode IPC_PRIVATE)
40     // et l'initialiser:
41
42     int semid = semget(IPC_PRIVATE, 1, 0641);
43     assert(semid != -1);
44
45     int semctl_ret = semctl(semid, 0, SETVAL, 0);
46     assert(semctl_ret != -1);
```

```

46 // Voyez-vous pourquoi il faut créer le sémaphore ici et non pas
47 // dans les fonctions un et deux ?
48
49 // Réponse: Le sémaphore ne sera pas le même entre un et deux
50 // à cause du fork() et donc on ne pourra pas mettre en oeuvre
51 // la précédence.
52
53 pid_t retFork;
54
55 retFork = fork();
56 assert(retFork != -1);
57 if (retFork != 0)
58     un(semid);
59 else
60     deux(semid);
61
62 // attente de la fin du fils
63 if (retFork != 0)
64     wait(NULL);
65
66 // ici il faut détruire le sémaphore
67 // chaque processus doit-il le détruire ?
68 // sinon qui doit s'en charger ?
69
70 //Réponse: Le sémaphore doit être détruit quand plus personne
71 //ne s'en sert, ie. quand le fils a fini. ie. que le père doit
72 //le détruire d'où cette condition (le père attend la fin de son
73 //fils, si on ne teste pas retFork == 0, le fils pourrait détruire
74 //le sémaphore).
75 if (retFork == 0)
76 {
77     int semctl_ret = semctl(semid, 1, IPC_RMID);
78     assert(semctl_ret != -1);
79 }
80
81 if (retFork != 0)
82     printf("Fin de un (%d)\n", getpid());
83 else
84     printf("Fin de deux (%d)\n", getpid());
85
86 return EXIT_SUCCESS;
87 }
```

**Listing 15.1** – Code source implantant la précédence entre un père et son fils

Explications :

### Modélisation de la précédence

- **Objectif** : un processus  $P_1$  doit s'exécuter après l'exécution partielle ou totale d'un processus  $P_0$
- **Principe** :
  - un sémaphore  $GO\_P1$  initialisé à 0 ;
  - lorsque le point d'exécution voulu de  $P_0$  est atteint, il l'indique en vendant le sémaphore  $GO\_P1$
  - lorsque le point d'exécution voulu de  $P_1$  est atteint, il attend l'autorisation de continuer en appelant  $P(GO\_P1)$

$P_0$

```
debut
  Code avant point de précédence
  V(GO_P1)
  Suite du code de P0
fin
```

$P_1$

```
debut
  Début de P1
  P(GO_P1)
  Code après point de précédence
fin
```

## Sémaphore et processus indépendants

### Question :

Le but est le même que l'exercice précédent, mais les deux processus faisant une précédence ne sont pas issus directement d'un fork. Il y a deux programmes exo5 un.c et exo5 deux.c dont les squelettes sont fournis. exo5 un.c doit afficher son message en premier et donc exo5 deux.c en dernier. Cela amène deux remarques :

- pour la clé il n'est pas possible de choisir IPC PRIVATE,
- un des deux processus devra créer le sémaphore et l'autre uniquement le récupérer sans le créer.

Pour le second point, nous allons le régler violemment : c'est exo5 un.c qui le crée, et exo5 deux.c qui le récupère en faisant l'hypothèse qu'il est déjà créé. Donc lors des tests, il faudra lancer exo5 un en premier. Il faut que le premier paramètre de semget soit le même dans les deux programmes. Pour cette première version de l'exercice vous le choisissez et l'écrivez en dur dans le code (par exemple 78624).

### Réponse :

```
1 #ifndef EXO5_COMM
2 #define EXO5_COMM
3
4 // clé choisie en dur pour l'identification du sémaphore
5 #define MA_CLE 78624
6
7 #endif
```

Listing 15.2 – exo5\_comm.h

```
1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 // ajouter ici les include nécessaires
10
11 #include <sys/ipc.h>
12 #include <sys/sem.h>
13
14 #include "exo5_comm.h"
15
16
17 int main()
18 {
19     // il faut créer le sémaphore avec la clé fournie dans exo5_comm.h
20     // il faut être rigoureux : si le sémaphore existe déjà,
21     // c'est une erreur et il ne faut pas oublier de l'initialiser :
22     int semid = semget(MA_CLE, 1, 0641 | IPC_CREAT | IPC_EXCL);
23     assert(semid != -1);
24
25     int semctl_ret = semctl(semid, 0, SETVAL, 0);
26     assert(semctl_ret != -1);
27
28 }
```

```

29 // une longue attente pour qu'on ait le temps de lancer exo5_deux
30 // et pour être sûr que ce dernier attend
31 sleep(5);
32 printf("Processus un %d ! (normalement je m'affiche en premier)\n",
33         getpid());
34 sleep(1);
35
36 // ici il faut débloquer le processus "deux"
37 struct sembuf semb = {0, 1, 0};
38 int semop_ret = semop(semid, &semb, 1);
39 assert(semop_ret != -1);
40
41
42 // Qui doit détruire le sémaphore ? un ou deux ?
43 // Rappelez-vous que lorsqu'on détruit un sémaphore, il faut
44 // être certain qu'aucun autre processus ne l'utilise.
45
46 //Réponse: Le processus "deux" se sert encore du sémaphore
47 //pour se débloquer. Si on le détruit ici on prend le risque
48 //que "un" détruise le sémaphore avant que "deux" n'ait eu
49 //le temps de s'en servir.
50
51
52 printf("Fin de un\n");
53
54 return EXIT_SUCCESS;
55 }
```

**Listing 15.3 – exo5\_un.c**

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 // ajouter ici les include nécessaires
10 #include <sys/ipc.h>
11 #include <sys/sem.h>
12
13 #include "exo5_comm.h"
14
15
16 int main()
17 {
18     // il faut récupérer le sémaphore avec la clé fournie dans exo5_comm.h
19     // il faut être rigoureux : le sémaphore doit déjà exister sinon
20     // c'est une erreur
21     // faut-il initialiser le sémaphore ?
22     int semid = semget(MA_CLE, 1, IPC_EXCL);
23     assert(semid != -1);
24
25
26     // ici il faut attendre le déblocage venant de "un"
```

```

27 struct sembuf semb = {0, -1, 0};
28 int semop_ret = semop(semid, &semb, 1);
29 assert(semop_ret != -1);
30
31
32 // affichage venant après celui de un
33 printf("Processus deux %d ! (normalement je m'affiche en dernier)\n",
34 getpid());
35
36 // Qui doit détruire le sémaphore ? un ou deux ?
37 // Rappelez-vous que lorsqu'on détruit un sémaphore, il faut
38 // être certain qu'aucun autre processus ne l'utilise.
39
40 //Réponse: Comme expliqué dans le code source de "un", "deux" est
41 //le dernier à se servir du sémaphore pour se débloquer. Donc il est
42 //celui qui doit le détruire.
43 semctl(semid, 0, IPC_RMID);
44
45
46 printf("Fin de deux\n");
47
48 return EXIT_SUCCESS;
49 }
```

**Listing 15.4 – exo5\_deux.c**

## Sémaphore et processus indépendants (version 2)

### Point de cours

Lors de la création d'un sémaphore, il faut lui associer une clé. Et comme le sémaphore est *a priori* visible par tous les autres processus (en fonction des droits tout de même), tout autre processus désirant manipuler le même sémaphore doit tout simplement utiliser la même clé.

Le problème est double :

- comment choisir la valeur de la clé ?
- comment tous les processus manipulant le sémaphore connaissent la clé ?

Le seconde point n'est pas un problème, il suffit par exemple de mettre sa valeur dans un *.h* commun<sup>a</sup>.

Le premier point est délicat car on ne peut pas choisir la valeur de manière arbitraire. En effet

souvenez-vous qu'un sémaphore n'est pas local à un utilisateur, mais global à la machine. Supposons que l'on choisisse la valeur 78623. Et bien on court le risque qu'un autre utilisateur de la machine choisisse la même clé<sup>b</sup>.

C'est alors la catastrophe car notre programme (ou celui de l'autre utilisateur) pourrait ne pas fonctionner ; ou pire ils pourraient se perturber l'un l'autre.

Il existe une fonction (*ftok*) dont le but est de générer des clés uniques. Elle prend en entrée un nom de fichier présent sur le système et renvoie une clé différente pour chaque fichier<sup>c</sup>.

La fonction est un peu plus souple. Elle prend un second paramètre qui permet de générer jusqu'à 255 clés différentes à partir du même fichier.

a. ou dégueux : mettre sa valeur en dur dans les programmes.

b. C'est certes peu probable, mais il n'est pas possible de prendre le risque.

c. La fonction utilise une partie de l'i-node du fichier, et donc les collisions sont en réalité possibles, cf. manuel.

### Question :

Reprenez l'exercice précédent, mais avec une clé générée par *ftok*. Les squelettes de exo6 un.c et exo6 deux.c sont fournis.

### Réponse :

```
1 #ifndef EXO6_COMM
2 #define EXO6_COMM
3
4 // fichier (qui doit être accessible) choisi pour l'identification du sé
5 // maphore
6 #define MON_FICHIER "exo6_comm.h"
7
8 // identifiant pour le deuxième paramètre de ftok
9 #define PROJ_ID 5
10#endif
```

Listing 15.5 – exo6\_comm.h

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 // ajouter ici les include nécessaires
10 #include <sys/ipc.h>
11 #include <sys/sem.h>
12
13 #include "exo6_comm.h"
14
15
16 int main()
17 {
18     // Il faut créer le sémaphore avec une clé générée par ftok et le
19     // le fichier fourni dans exo6_comm.h
20     // Il faut être rigoureux : si le sémaphore existe déjà, c'est une
21     // erreur
22     // et il ne faut pas oublier de l'initialiser
23     key_t mykey = ftok(MON_FICHIER, PROJ_ID);
24     int semid = semget(mykey, 1, 0641 | IPC_CREAT | IPC_EXCL);
25     assert(semid != -1);
26
27     int semctl_ret = semctl(semid, 0, SETVAL, 0);
28     assert(semctl_ret != -1);
29
30     // une longue attente pour qu'on ait le temps de lancer exo5_deux
31     // et pour être sûr que ce dernier attend
32     sleep(5);
33     printf("Processus un %d ! (normalement je m'affiche en premier)\n",
34           getpid());
35     sleep(1);
36
37     // ici il faut débloquer le processus "deux"
38     struct sembuf semb = {0, 1, 0};
39     int semop_ret = semop(semid, &semb, 1);
40     assert(semop_ret != -1);
41
42     // Qui doit détruire le sémaphore ? un ou deux ?
43     // Rappelez-vous que lorsqu'on détruit un sémaphore, il faut
44     // être certain qu'aucun autre processus ne l'utilise.
45
46     //Réponse: "deux" doit encore se servir du sémaphore pour se débloquer
47     //donc c'est "deux" qui doit le détruire et non "un".
48
49     printf("Fin de un\n");
50
51     return EXIT_SUCCESS;
52 }
```

Listing 15.6 – exo6\_un.c

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 // ajouter ici les include nécessaires
10 #include <sys/ipc.h>
11 #include <sys/sem.h>
12
13
14 #include "exo6_comm.h"
15
16
17 int main()
18 {
19     // Il faut récupérer le sémaphore avec une clé générée par ftok et le
20     // le fichier fourni dans exo6_comm.h
21     // Il faut être rigoureux : le sémaphore doit déjà exister sinon
22     // c'est une erreur
23     // faut-il initialiser le sémaphore ?
24     key_t mykey = ftok(MON_FICHIER, PROJ_ID);
25     int semid = semget(mykey, 1, IPC_EXCL);
26     assert(semid != -1);
27
28     // ici il faut attendre le déblocage venant de "un"
29     struct sembuf semb = {0, -1, 0};
30     int semop_ret = semop(semid, &semb, 1);
31     assert(semop_ret != -1);
32
33
34     // affichage venant après celui de un
35     printf("Processus deux %d ! (normalement je m'affiche en dernier)\n",
36           getpid());
37
38     // Qui doit détruire le sémaphore ? un ou deux ?
39     // Rappelez-vous que lorsqu'on détruit un sémaphore, il faut
40     // être certain qu'aucun autre processus ne l'utilise.
41
42     //Réponse: "deux" est le dernier processus à se servir du sémaphore.
43     //Il est donc celui qui peut le supprimer
44
45     int semctl_ret = semctl(semid, 0, IPC_RMID);
46     assert(semctl_ret != -1);
47
48     printf("Fin de deux\n");
49
50     return EXIT_SUCCESS;
51 }
```

**Listing 15.7 – exo6\_deux.c**

## Sémaphore et barrière de synchronisation

**Question :** Le but est ici que plusieurs processus s'attendent mutuellement au milieu de leur code avant de continuer.

Voici le pseudo-code des processus :

```
début
je fais la 1re partie du code
...
j'attends ici que tous les autres arrivent au même point
// si je suis là, alors tous les autres aussi (ou plus loin)
je fais la 2e partie du code
...
fin
```

Techniquement il s'agit d'utiliser l'opération 0 (zéro) de la fonction semop. Le squelette des processus devant se synchroniser est fourni : exo7 worker.c.

Il reste toujours le problème de la création et de la destruction du sémaphore. Pour cela un programme à part se charge de ces opérations (squelette exo7 master.c). Il faut le lancer en premier en indiquant combien de workers vont être lancés ; et il faudra lui indiquer de détruire le sémaphore lorsque les workers auront terminé.

Il faudra une console pour le master et une console par worker.

**Réponse :**

```
1 #ifndef EXO7_COMM
2 #define EXO7_COMM
3
4 // fichier (qui doit être accessible) choisi pour l'identification du sé
5 // maphore
6 #define MON_FICHIER "exo7_comm.h"
7
8 // identifiant pour le deuxième paramètre de ftok
9 #define PROJ_ID 5
10
11 #endif
```

Listing 15.8 – exo7\_comm.h

```
1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 #include <sys/ipc.h>
10 #include <sys/sem.h>
11
12 #include "exo7_comm.h"
13
14 const int MIN_WORKERS = 1;
```

```

15 const int MAX_WORKERS = 7;
16
17 ///////////////////////////////////////////////////////////////////
18 static void usage(const char *exeName, const char *message)
19 {
20     fprintf(stderr, "usage : %s <nbre workers>\n", exeName);
21     fprintf(stderr, "           nombre de workers compris entre %d et %d\n",
22             MIN_WORKERS, MAX_WORKERS);
23     fprintf(stderr, "           le programme ne vérifie pas que le paramètre "
24             "'est un entier correctement constitué.\n");
25     if (message != NULL)
26         fprintf(stderr, "message : %s\n", message);
27     exit(EXIT_FAILURE);
28 }
29
30
31 ///////////////////////////////////////////////////////////////////
32 // Il faut créer le sémaphore avec une clé générée par ftok et le
33 // le fichier fourni dans exo7_comm.h
34 // Il faut être rigoureux : si le sémaphore existe déjà, c'est une erreur
35 // et il ne faut pas oublier de l'initialiser en fonction du nombre
36 // de workers
37 static int my_semget(int nbreWorkers)
38 {
39     key_t mykey = ftok(MON_FICHIER, PROJ_ID);
40     assert(mykey != -1);
41
42     int semid = semget(mykey, 1, 0641 | IPC_CREAT | IPC_EXCL);
43     assert(semid != -1);
44
45     int semctl_ret = semctl(semid, 0, SETVAL, nbreWorkers);
46     assert(semctl_ret != -1);
47
48     return semid;
49 }
50
51 ///////////////////////////////////////////////////////////////////
52 // destruction du sémaphore
53 static void my_destroy(int semId)
54 {
55     int semctl_ret = semctl(semId, 0, IPC_RMID);
56     assert(semctl_ret != -1);
57 }
58
59
60 ///////////////////////////////////////////////////////////////////
61 int main(int argc, char * argv[])
62 {
63     if (argc != 2)
64         usage(argv[0], "nombre de paramètres incorrect");
65
66     int nbreWorkers;
67     int semId;
68
69     nbreWorkers = strtol(argv[1], NULL, 10);
70     if ((nbreWorkers < MIN_WORKERS) || (nbreWorkers > MAX_WORKERS))

```

```

71     usage(argv[0], "nbre workers incorrect");
72
73 // création du sémaphore
74 semId = my_semget(nbreWorkers);
75
76 printf("Le sémaphore est créé et vous devez lancer %d workers\n",
77        nbreWorkers);
78 printf("Lorsque tous les workers ont terminé, appuyez sur <Entrée>\n");
79
80 printf("—> ");
81 getchar();
82
83 // destruction du sémaphore
84 // si vous êtes curieux, faites-le pendant que les workers l'utilisent
85 my_destroy(semId);
86
87 printf("Fin du master\n");
88
89 return EXIT_SUCCESS;
90 }
```

**Listing 15.9 – exo7\_master.c**

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 #include <sys/IPC.h>
10 #include <sys/sem.h>
11
12 #include "exo7_comm.h"
13
14 const int MIN_SECONDES = 3;
15 const int MAX_SECONDES = 9;
16
17
18 //////////////////////////////////////////////////////////////////
19 // Il faut récupérer le sémaphore avec une clé générée par ftok et le
20 // le fichier fourni dans exo7_comm.h
21 // Il faut être rigoureux : le sémaphore doit déjà exister sinon
22 // c'est une erreur
23 static int my_semget()
24 {
25     key_t mykey = ftok(MON_FICHIER, PROJ_ID);
26     assert(mykey != -1);
27
28     int semid = semget(mykey, 1, IPC_EXCL);
29     assert(semid != -1);
30
31     return semid;
32 }
```

```

34 // Le code doit attendre tous les autres workers
35 // static void my_barriere(int semId)
36 {
37     struct sembuf semb1 = {0, -1, 0};
38     semop(semId, &semb1, 1);
39
40     struct sembuf semb2 = {0, 0, 0};
41     semop(semId, &semb2, 1);
42 }
43
44
45
46 // int main()
47 {
48     int tempsAttente;
49     int semId;
50
51     // récupération du sémaphore
52     semId = my_semget();
53
54     // simule la 1re partie du code
55     srand(getpid());
56     tempsAttente = MIN_SECONDES + rand() % (MAX_SECONDES - MIN_SECONDES + 1);
57     printf("1re partie : je bosse pendant %d seconde(s)\n", tempsAttente);
58     printf("           Zzz...\n");
59     sleep(tempsAttente);
60     printf("           j'ai fini.\n");
61
62     printf("\n");
63     printf("J'attends les autres\n");
64
65     // appel de la fonction implémentant la barrière
66     my_barriere(semId);
67
68     printf("\n");
69     printf("Tout le monde a franchi la barrière\n");
70     printf("\n");
71
72     // simule la 2me partie du code
73     srand(getpid());
74     tempsAttente = 1 + rand() % MIN_SECONDES;
75     printf("2me partie : je bosse pendant %d seconde(s)\n", tempsAttente);
76     printf("           Zzz...\n");
77     sleep(tempsAttente);
78     printf("           j'ai fini.\n");
79
80     printf("\n");
81     printf("Fin du worker\n");
82
83
84     return EXIT_SUCCESS;
85 }
```

**Listing 15.10 – exo7\_worker.c**

# Mémoire partagée

## Point de cours

Deux (ou plus) processus lourds indépendants ou issus directement d'un *fork* ont par définition des espaces mémoire indépendants (au contraire des threads).

Il est possible malgré tout d'utiliser une zone partagée (ou encore segment partagé) via des primitives IPC.

La démarche est la suivante :

- création ou récupération de l'objet IPC : fonction *shmget*
- déclaration d'un pointeur qui pointe sur cette zone : fonction *shmat*
- travail avec la zone
- détachement du pointeur : fonction *shmdt*
- éventuellement destruction du segment : fonction *shmctl*

Pour avoir les liste des segments partagés présents sur la machine :

```
$ ipcs -m
```

## Question :

Écrivez deux programmes. Le premier :

- crée le segment
- écrit une chaîne de caractères
- détruit le segment

Le second :

- récupère le segment
- lit la chaîne -
- ne détruit pas le segment puisque c'est le premier processus qui le fait

Attention à faire les opérations dans le bon ordre (synchronisation ?). Par exemple il ne faut pas que le second processus lise le segment avant que le premier n'ait écrit dedans.

## Réponse :

```
1 #ifndef EXO8_COMM
2 #define EXO8_COMM
3
4 // fichier (qui doit être accessible) choisi pour l'identification du sé
5 // maphore
6 #define MON_FICHIER "exo8_comm.h"
7
8 // identifiants pour le deuxième paramètre de ftok
9 #define PROJ_ID_SEM_ECRITURE 5
10 #define PROJ_ID_SEM_FIN 6
11 #define PROJ_ID_SHM 5
12
13 // taille segment (surdimensionné)
14 #define SHM_TAILLE 100*sizeof(char)
15 #endif
```

Listing 15.11 – exo8\_comm.h

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <assert.h>
8 #include <sys/types.h>
9 #include <unistd.h>
10 #include <sys/IPC.h>
11 #include <sys/sem.h>
12 #include <sys/shm.h>
13
14 #include "exo8_comm.h"
15
16 // création d'un sémaphore IPC
17 static int my_semget(int projectId, int initVal)
18 {
19     key_t mykey = ftok(MON_FICHIER, projectId);
20     assert(mykey != -1);
21
22     int semid = semget(mykey, 1, 0641 | IPC_CREAT | IPC_EXCL);
23     assert(semid != -1);
24
25     int semctl_ret = semctl(semid, 0, SETVAL, initVal);
26     assert(semctl_ret != -1);
27
28     return semid;
29 }
30
31 // -1 sur un sémaphore
32 static void prendre(int semId)
33 {
34     struct sembuf semb = {0, -1, 0};
35     int semop_ret = semop(semId, &semb, 1);
36     assert(semop_ret != -1);
37 }
38
39 // +1 sur un sémaphore
40 static void vendre(int semId)
41 {
42     struct sembuf semb = {0, 1, 0};
43     int semop_ret = semop(semId, &semb, 1);
44     assert(semop_ret != -1);
45 }
46
47 // création d'un segment
48 static int my_shmget()
49 {
50     key_t shmkey = ftok(MON_FICHIER, PROJ_ID_SHM);
51     assert(shmkey != -1);
52
53     int shmid = shmget(shmkey, SHM_TAILLE, 0641|IPC_CREAT|IPC_EXCL);
54     assert(shmid != -1);
55 }
```

```

56     return shmid;
57 }
58
59 // attachement sur un segment
60 static char * my_shmat(int shmId)
61 {
62     char* shmat_ret = (char*) shmat(shmId, NULL, 0);
63     assert(shmat_ret != (void *) -1);
64
65     return shmat_ret;
66 }
67
68 // détachement d'un segment
69 void my_shmdt(void *pt)
70 {
71     int shmdt_ret = shmdt(pt);
72     assert(shmdt_ret != -1);
73 }
74
75 // suppression des sémaphores et du segment
76 static void my_destroyAll(int sem1, int sem2, int shm)
77 {
78     // Détruire les deux sémaphores et le segment
79     int semctl_ret1 = semctl(sem1, 0, IPC_RMID);
80     assert(semctl_ret1 != -1);
81
82     int semctl_ret2 = semctl(sem2, 1, IPC_RMID);
83     assert(semctl_ret2 != -1);
84
85     int shmctl_ret = shmctl(shm, IPC_RMID, NULL);
86     assert(shmctl_ret != -1);
87 }
88
89
90 int main()
91 {
92     const char * const chaine = "Bonjour monde !";
93     int semIdEcriture;
94     int semIdFin;
95     int shmId;
96     char * ptShm = NULL;
97
98     // création sémaphores
99     semIdEcriture = my_semget( PROJ_ID_SEM_ECRITURE, 0);
100    semIdFin = my_semget( PROJ_ID_SEM_FIN, 0);
101
102    // création segment (rappel : on suppose le segment assez grand)
103    shmId = my_shmget();
104    // et un pointeur sur le segment
105    ptShm = my_shmat(shmId);
106
107    // on écrit avec une tempo
108    sleep(3);
109    printf("Je vais écrire la chaine\n");
110    strcpy(ptShm, chaine);
111    sleep(1);

```

```

112 printf("Chaine écrite\n");
113 // on débloque deux
114 vendre(semIdEcriture);
115
116 // on attend que deux n'ait plus besoin du segment (et des sémaphore)
117 prendre(semIdFin);
118
119 // on détache le pointeur
120 my_shmdt(ptShm);
121
122 // on détruit tout le monde
123 my_destroyAll(semIdEcriture, semIdFin, shmId);
124
125
126
127 printf("Fin de un\n");
128
129 return EXIT_SUCCESS;
130 }
```

**Listing 15.12 – exo8\_un.c**

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 #include <sys/IPC.h>
10 #include <sys/sem.h>
11 #include <sys/shm.h>
12
13 #include "exo8_comm.h"
14
15
16 // récupération d'un sémaphore IPC
17 static int my_semget(int projectId)
18 {
19     key_t mykey = ftok(MON_FICHIER, projectId);
20     assert(mykey != -1);
21
22     int semid = semget(mykey, 1, IPC_EXCL);
23     assert(semid != -1);
24
25     return semid;
26 }
27
28 // -1 sur un sémaphore
29 static void prendre(int semId)
30 {
31     struct sembuf semb = {0, -1, 0};
32     int semop_ret = semop(semId, &semb, 1);
33     assert(semop_ret != -1);
34 }
```

```

35 // +1 sur un sémaphore
36 static void vendre(int semId)
37 {
38     struct sembuf semb = {0, 1, 0};
39     int semop_ret = semop(semId, &semb, 1);
40     assert (semop_ret != -1);
41 }
42
43
44 // récupération d'un segment
45 static int my_shmget()
46 {
47     key_t shmkey = ftok(MON_FICHIER, PROJ_ID_SHM);
48     assert (shmkey != -1);
49
50     int shmid = shmget(shmkey, SHM_TAILLE, IPC_EXCL);
51     assert (shmget_ret != -1);
52
53     return shmid;
54 }
55
56 // attachement sur un segment
57 static char * my_shmat(int shmId)
58 {
59     char* shmat_ret = (char*) shmat(shmId, NULL, 0);
60     assert (shmat_ret != (void *) -1);
61
62     return shmat_ret;
63 }
64
65 // détachement d'un segment
66 void my_shmdt(void *pt)
67 {
68     int shmdt_ret = shmdt(pt);
69     assert (shmdt_ret != -1);
70 }
71
72
73 int main()
74 {
75     int semIdEcriture;
76     int semIdFin;
77     int shmid;
78     char * ptShm = NULL;
79
80     // récupération sémaphores
81     semIdEcriture = my_semget( PROJ_ID_SEM_ECRITURE);
82     semIdFin = my_semget(PROJ_ID_SEM_FIN);
83
84     // création segment (rappel : on suppose le segment assez grand)
85     shmid = my_shmget();
86     // et un pointeur sur le segment (autant le mettre en read only)
87     ptShm = my_shmat(shmid);
88
89     // on attend l'autorisation de un
90     prendre(semIdEcriture);

```

```
91 // On affiche la chaîne
92 printf("chaîne : %s\n", ptShm);
93
94 // on détache le pointeur
95 my_shmdt(ptShm);
96
97 // on dit à un qu'il peut tout détruire
98 vendre(semIdFin);
99
100 // et donc rien à détruire
101
102 printf("Fin de deux\n");
103
104 return EXIT_SUCCESS;
105
106 }
```

**Listing 15.13** – exo8\_deux.c

## File de messages

### Point de cours

Le troisième type d'objet IPC est la file de messages. Par rapport à un tube, le fonctionnement est plus évolué ; notamment les messages sont typés et deux messages de types différents peuvent avoir des priorités différentes.

La démarche est la suivante :

- création ou récupération de l'objet IPC : fonction *msgget*
- envois de messages : fonction *msgsnd*
- et/ou réceptions de messages : fonction *msgrcv*
- éventuellement destruction de la file : fonction *msgctl*

Pour avoir la liste des files présentes sur la machine :

```
$ ipcs -q
```

### Question :

Écrivez trois programmes.

Le premier :

- crée la file
- y envoie deux suites de valeurs :
  - une d'entiers terminée par -1
  - une de caractères terminée par '.'
- c'est le type du message qui indique à quelle suite appartient une valeur. Les deux suites s'entrechoquent et ont des longueurs non connues par les deux processus récepteurs.
- détruit la file

Le second :

- récupère la file
- lit la suite d'entiers associés au premier type et en fait la somme
- ne détruit pas la file puisque c'est le premier processus qui le fait

Le troisième :

- récupère la file
- lit la suite de caractères associés au second type et les affiche
- ne détruit pas la file puisque c'est le premier processus qui le fait

Attention à faire les opérations dans le bon ordre (synchronisation ?). Par exemple il ne faut pas que le premier processus détruise la file si elle est encore utilisée par les récepteurs.

Réponse :

```
1 #ifndef EXO9_COMM
2 #define EXO9_COMM
3
4 /**************************************************************************
5 * données pour ftok
6 **************************************************************************/
7 // fichier (qui doit être accessible) choisi pour l'identification de la
8 // file
9 #define MON_FICHIER "exo9_comm.h"
10
11 // identifiants pour le deuxième paramètre de ftok
12 #define PROJ_ID 5
13
14 /**************************************************************************
15 * données pour les messages d'entiers
16 **************************************************************************/
17 // type struct
18 typedef struct
19 {
20     long mtype;
21     int val;
22 } MsgInt;
23
24 // taille (pour shmsnd)
25 #define MSG_INT_TAILLE ( sizeof(MsgInt) - sizeof(long) )
26
27 // valeur du type de messages
28 #define MSG_INT_TYPE 5
29
30 /**************************************************************************
31 * données pour les messages de char
32 **************************************************************************/
33 // type struct
34 typedef struct
35 {
36     long mtype;
37     char val;
38 } MsgChar;
39
40 #define MSG_CHAR_TAILLE ( sizeof(MsgChar) - sizeof(long) )
41
42 // valeur du type de messages
43 #define MSG_CHAR_TYPE 9
44
45 #endif
```

Listing 15.14 – exo9\_comm.h

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <unistd.h>
8 #include <sys/types.h>
9 #include <sys/ipc.h>
10 #include <sys/msg.h>
11
12 #include "exo9_comm.h"
13
14 // création d'une file IPC
15 static int my_msgget()
16 {
17     // Création de la file:
18     key_t mykey = ftok(MON_FICHIER, PROJ_ID);
19     assert(mykey != -1);
20
21     int msgid = msgget(mykey, 0641|IPC_CREAT|IPC_EXCL);
22     assert(msgid != -1);
23
24     return msgid;    // il faut retourner l'id de la file
25 }
26
27 // envoi d'un entier
28 void my_msghsndInt(int msgId, int v)
29 {
30     // Envoyer l'entier v avec le type MSG_INT_TYPE:
31     MsgInt msg = {MSG_INT_TYPE, v};
32     int msg_ret = msgsnd(msgId, &msg, MSG_INT_TAILLE, 0);
33     assert(msg_ret != -1);
34 }
35
36 // envoi d'un caractère
37 void my_msghsndChar(int msgId, char v)
38 {
39     // Envoyer le caractère v avec le type MSG_CHAR_TYPE:
40     MsgChar msg = {MSG_CHAR_TYPE, v};
41     int msg_ret = msgsnd(msgId, &msg, MSG_CHAR_TAILLE, 0);
42     assert(msg_ret != -1);
43 }
44
45 // suppression de la file
46 static void my_msgDestroy(int msgId)
47 {
48     // Description de la file:
49     int msgctl_ret = msgctl(msgId, IPC_RMID, NULL);
50     assert(msgctl_ret != -1);
51 }
52
53
54 int main()
55 {

```

```

56 int msgId;
57
58 // création file
59 msgId = my_msgget();
60 sleep(1);
61
62 // on envoie différentes valeurs
63 my_msghsndInt(msgId, 1);
64 my_msghsndInt(msgId, 2);
65 my_msghsndChar(msgId, 'A');
66 sleep(1);
67 my_msghsndInt(msgId, 3);
68 my_msghsndInt(msgId, 4);
69 my_msghsndChar(msgId, 'B');
70 my_msghsndChar(msgId, 'C');
71 sleep(2);
72 my_msghsndChar(msgId, 'D');
73 my_msghsndInt(msgId, 5);
74 my_msghsndChar(msgId, '.'); // fin suite char
75 my_msghsndInt(msgId, 6);
76 my_msghsndInt(msgId, 7);
77 sleep(1);
78 my_msghsndInt(msgId, -1); // fin suite int
79
80 // on détruit la file
81 // Attention, il faudrait être sûr que les deux autres
82 // processus n'utilise plus la file, et pour cela
83 // il faudrait un sémaphore
84 // On fait plus simple : c'est l'utilisateur qui donne l'ordre
85 // de destruction
86 printf("Appuyez sur \"Entrée\" pour détruire la file.\n");
87 printf("—> ");
88 getchar();
89
90 my_msgDestroy(msgId);
91
92 printf("Fin de l'émetteur\n");
93
94 return EXIT_SUCCESS;
95 }
```

**Listing 15.15 – exo9\_sender.c**

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <stdbool.h>
7 #include <assert.h>
8 #include <sys/types.h>
9 #include <sys/ipc.h>
10 #include <sys/msg.h>
11
12 #include "exo9_comm.h"
13
```

```

14 // récupérer la file IPC
15 static int my_msgget()
16 {
17     // Récupérer la file :
18     key_t mykey = ftok(MON_FICHIER, PROJ_ID);
19     assert (mykey != -1);
20
21     int msgid = msgget(mykey, IPC_EXCL);
22     assert (msgid != -1);
23
24     return msgid;    // il faut retourner l'id de la file
25 }
26
27 // lecture d'un message
28 int my_msgrvcInt(int msgId)
29 {
30     // Recevoir un message de type MSG_INT_TYPE:
31     MsgInt msg = {MSG_INT_TYPE, 0};
32     int msg_ret = msgrcv(msgId, (int *) &msg, MSG_INT_TAILLE, MSG_INT_TYPE,
33     0);
34     assert (msg_ret != -1);
35
36     return msg.val;    // il faut retourner l'entier lu
37 }
38
39 int main()
40 {
41     int msgId;
42
43     // récupération de la file
44     msgId = my_msgget();
45
46     // réception des valeurs
47     while (true)
48     {
49         int v = my_msgrvcInt(msgId);
50         if (v == -1)
51             break;
52         printf("J'ai reçu %d\n", v);
53     }
54
55     // pas de destruction de la file
56
57     printf("Fin du récepteur d'entiers\n");
58
59     return EXIT_SUCCESS;
60 }
```

**Listing 15.16 – exo9\_reader\_int.c.c**

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <stdbool.h>
7 #include <assert.h>
8 #include <sys/types.h>
9 #include <sys/ipc.h>
10 #include <sys/msg.h>
11
12 #include "exo9_comm.h"
13
14 // Récupérer la file IPC:
15 static int my_msgget()
16 {
17     // Récupérer la file :
18     key_t mykey = ftok(MON_FICHIER, PROJ_ID);
19     assert(mykey != -1);
20
21     int msgid = msgget(mykey, IPC_EXCL);
22     assert(msgid != -1);
23
24     return msgid;    // il faut retourner l'id de la file
25 }
26
27 // Lecture d'un message:
28 char my_msgrvcChar(int msgId)
29 {
30     // Recevoir un message de type MSG_CHAR_TYPE:
31     MsgInt msg = {MSG_CHAR_TYPE, 0};
32     int msg_ret = msgrcv(msgId, (char *) &msg, MSG_CHAR_TAILLE,
33     MSG_CHAR_TYPE, 0);
34     assert(msg_ret != -1);
35
36     return msg.val; // il faut retourner le caractère lu
37 }
38
39 int main()
40 {
41     int msgId;
42
43     // récupération de la file
44     msgId = my_msgget();
45
46     // réception des valeurs
47     while (true)
48     {
49         char v = my_msgrvcChar(msgId);
50         if (v == '.')
51             break;
52         printf("J'ai reçu %c\n", v);
53     }
54 }
```

```
55 // pas de destruction de la file
56
57 printf("Fin du récepteur de char\n");
58
59 return EXIT_SUCCESS;
60 }
```

**Listing 15.17** – exo9\_reader\_char.c

## Paquetage de sémaphore

**Question :** Comme l'utilisation des sémaphores est assez technique, il peut être intéressant de faire un module qui masque l'implémentation.

Programmez un tel module dont le squelette vous est fourni :

- Semaphore.h : à ne pas modifier
- Semaphore.c : à compléter

Reprenez l'exercice 7 en utilisant le module plutôt que de programmer directement les sémaphores.

**Réponse :**

```
1 #ifndef SEMAPHORE_H
2 #define SEMAPHORE_H
3
4 struct SemaphoreP;
5 typedef struct SemaphoreP *Semaphore;
6 typedef const struct SemaphoreP *constSemaphore;
7
8 /**
9  * constructeurs et destructeur
10 * surtout ne pas oublier de détruire le sémaphore pour ne pas
11 * saturer le système
12 * les sémaphores sont initialisés à 0 par défaut
13 */
14 // création sémaphore IPC_PRIVATE
15 Semaphore sema_creerPrivate(int nbSem, int droits);
16 // création : le sémaphore ne doit pas exister sur le système
17 Semaphore sema_creerNouveau(const char *filename, int idProj, int droits,
18                             int nbSem);
19 // récupérer un sémaphore existant
20 Semaphore sema_creerExistant(const char *filename, int idProj, int nbSem);
21 // destructeur sans suppression sur le système
22 void sema_detruireSansSuppression(Semaphore *self);
23 // destructeur avec suppression sur le système
24 void sema_detruireAvecSuppression(Semaphore *self);
25
26 /**
27  * initialisation , accesseurs
28 */
29 unsigned short sema_getVal(constSemaphore self, int pos);
30 void sema_setVal(Semaphore self, int pos, unsigned short val);
31 void sema_getVals(constSemaphore self, int nbSem, unsigned short vals[]);
32 void sema_setVals(Semaphore self, int nbSem, const unsigned short vals[]);
33
34 /**
35  * opérations basiques
36 */
37 // tentative de faire -1 sur le sémaphore
38 void sema_prendre(Semaphore self, int pos);
39
40 // +1 sur le sémaphore
41 void sema_vendre(Semaphore self, int pos);
```

```

43 // attente que le sémaphore passe à 0
44 void sema_attendre(Semaphore self, int pos);
45
46 /**
47 * opérations à peine moins basiques
48 */
49 // équivalent de prendre avec une valeur pouvant être > 1
50 // delta est bien > 0, et on essaie de retrancher delta au sémaphore
51 void sema_diminue(Semaphore self, int pos, int delta);
52
53 // équivalent de vendre avec une valeur pouvant être > 1
54 // on ajoute delta au sémaphore
55 void sema_augmente(Semaphore self, int pos, int delta);
56
57 // équivaut à :
58 //   si delta < 0 : sema_diminue(self, -delta);
59 //   si delta = 0 : sema_attendre(self);
60 //   si delta > 0 : sema_augmente(self, delta);
61 void sema_modifie(Semaphore self, int pos, int delta);
62
63
64 #endif

```

**Listing 15.18** – Semaphore.h

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdlib.h>
5 #include <stdbool.h>
6 #include <sys/types.h>
7 #include <sys/IPC.h>
8 #include <sys/sem.h>
9 #include <assert.h>
10
11 #include "myassert.h"
12
13 #include "Semaphore.h"
14
15 struct SemaphoreP
16 {
17     int nbSem;
18     int semId;
19 };
20
21 /**
22 * constructeurs, destructeurs
23 */
24
25 // pensez éventuellement à factoriser le code commun aux constructeurs
26
27 Semaphore sema_creerPrivate(int nbSem, int droits)
28 {
29     int semid = semget(IPC_PRIVATE, nbSem, droits | IPC_CREAT | IPC_EXCL);
30     assert(semid != -1);
31

```

```

32     Semaphore res = (Semaphore) malloc(sizeof(struct SemaphoreP));
33     res->nbSem = nbSem;
34     res->semId = semid;
35     return res;
36 }
37
38 Semaphore sema_creerNouveau(const char *filename, int idProj, int droits,
39                             int nbSem)
40 {
41     key_t mykey = ftok(filename, idProj);
42     assert(mykey != -1);
43
44     int semid = semget(mykey, nbSem, droits|IPC_CREAT|IPC_EXCL);
45     assert(semid != -1);
46
47     Semaphore res = (Semaphore) malloc(sizeof(struct SemaphoreP));
48     res->nbSem = nbSem;
49     res->semId = semid;
50     return res;
51 }
52
53 Semaphore sema_creerExistant(const char *filename, int idProj, int nbSem)
54 {
55     // Récupérer la file :
56     key_t mykey = ftok(filename, idProj);
57     assert(mykey != -1);
58
59     int semid = semget(mykey, nbSem, IPC_EXCL);
60     assert(semid != -1);
61
62     Semaphore res = (Semaphore) malloc(sizeof(struct SemaphoreP));
63     res->nbSem = nbSem;
64     res->semId = semid;
65     return res;
66 }
67
68 void sema_detruireSansSupression(Semaphore *self)
69 {
70     free(*self);
71     *self = NULL;
72 }
73
74 void sema_detruireAvecSupression(Semaphore *self)
75 {
76     int semctl_ret = semctl((*self)->semId, 0, IPC_RMID);
77     assert(semctl_ret != -1);
78     sema_detruireSansSupression(self);
79 }
80
81
82 /**
83 * initialisation, accesseurs
84 */
85
86 unsigned short sema_getVal(constSemaphore self, int pos)

```

```

87 {
88     myassert(( pos >= 0) && ( pos < self->nbSem) , "numero semaphore");
89
90     int semctl_ret = semctl(self->semId, pos, GETVAL);
91     assert(semctl_ret != -1);
92     return semctl_ret;
93 }
94
95 void sema_setVal(Semaphore self, int pos, unsigned short val)
96 {
97     myassert(( pos >= 0) && ( pos < self->nbSem) , "numero semaphore");
98
99     int semctl_ret = semctl(self->semId, pos, SETVAL, val);
100    assert(semctl_ret != -1);
101 }
102
103 void sema_getVals(constSemaphore self, int nbSem, unsigned short vals[])
104 {
105     myassert(nbSem == self->nbSem, "nombre semaphores");
106
107     for(int i = 0; i < nbSem; i++)
108         vals[i] = sema_getVal(self, i);
109 }
110
111 void sema_setVals(Semaphore self, int nbSem, const unsigned short vals[])
112 {
113     myassert(nbSem == self->nbSem, "nombre semaphores");
114
115     for(int i = 0; i < nbSem; i++)
116         sema_setVal(self, i, vals[i]);
117 }
118
119
120 /**
121 * opérations basiques
122 */
123
124 // pensez éventuellement à factoriser le code commun aux opérations
125
126 void sema_prendre(Semaphore self, int pos)
127 {
128     struct sembuf semb = {pos, -1, 0};
129     int semop_ret = semop(self->semId, &semb, 1);
130     assert(semop_ret != -1);
131 }
132
133 void sema_vendre(Semaphore self, int pos)
134 {
135     struct sembuf semb = {pos, 1, 0};
136     int semop_ret = semop(self->semId, &semb, 1);
137     assert(semop_ret != -1);
138 }
139
140 void sema_attendre(Semaphore self, int pos)
141 {
142     struct sembuf semb = {pos, 0, 0};

```

```

143     int semop_ret = semop(self->semId, &semb, 1);
144     assert(semop_ret != -1);
145 }
146
147
148 /**
149 * opérations à peine moins basiques
150 */
151 void sema_diminue(Semaphore self, int pos, int delta)
152 {
153     myassert(delta > 0, "delta doit etre > 0");
154
155     struct sembuf semb = {pos, -delta, 0};
156     int semop_ret = semop(self->semId, &semb, 1);
157     assert(semop_ret != -1);
158 }
159
160 void sema_augmente(Semaphore self, int pos, int delta)
161 {
162     myassert(delta > 0, "delta doit etre > 0");
163
164     struct sembuf semb = {pos, delta, 0};
165     int semop_ret = semop(self->semId, &semb, 1);
166     assert(semop_ret != -1);
167 }
168
169 void sema_modifie(Semaphore self, int pos, int delta)
170 {
171     if(delta < 0)
172     {
173         sema_diminue(self, pos, -delta);
174     }
175
176     else if(delta == 0)
177     {
178         sema_attendre(self, pos);
179     }
180
181     else
182     {
183         sema_augmente(self, pos, delta);
184     }
185 }
```

**Listing 15.19 – Semaphore.c**

```

1 #ifndef EXO10_COMM
2 #define EXO10_COMM
3
4 // fichier (qui doit être accessible) choisi pour l'identification du sé
5 // maphore
6 #define MON_FICHIER "exo10_comm.h"
7
8 // identifiant pour le deuxième paramètre de ftok
9 #define PROJ_ID 5
10
11 #endif

```

**Listing 15.20 – exo10\_comm.h**

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 #include <sys/IPC.h>
10 #include <sys/sem.h>
11
12 #include "Semaphore.h"
13 #include "exo10_comm.h"
14
15 const int MIN_WORKERS = 1;
16 const int MAX_WORKERS = 7;
17
18 //_____
19 static void usage(const char *exeName, const char *message)
20 {
21     fprintf(stderr, "usage : %s <nbre workers>\n", exeName);
22     fprintf(stderr, "           nombre de workers compris entre %d et %d\n",
23             MIN_WORKERS, MAX_WORKERS);
24     fprintf(stderr, "           le programme ne vérifie pas que le paramètre "
25             "'est un entier correctement constitué.\n");
26     if (message != NULL)
27         fprintf(stderr, "message : %s\n", message);
28     exit(EXIT_FAILURE);
29 }
30
31
32 //_____
33 // Il faut créer le sémaphore avec une clé générée par ftok et le
34 // le fichier fourni dans exo7_comm.h
35 // Il faut être rigoureux : si le sémaphore existe déjà, c'est une erreur
36 // et il ne faut pas oublier de l'initialiser en fonction du nombre
37 // de workers
38 static Semaphore my_semget(int nbreWorkers)
39 {
40     Semaphore s = sema_creerNouveau(MON_FICHIER, PROJ_ID, 0641, 1);
41     sema_setVal(s, 0, (unsigned short) nbreWorkers);

```

```

42     return s;
43 }
44
45 //-
46 // destruction du sémaphore
47 static void my_destroy(Semaphore *s)
48 {
49     sema_detruireAvecSupression(s);
50 }
51
52 //-
53 int main(int argc, char * argv[])
54 {
55     if (argc != 2)
56         usage(argv[0], "nombre de paramètres incorrect");
57
58     int nbreWorkers;
59     Semaphore sem;
60
61     nbreWorkers = strtol(argv[1], NULL, 10);
62     if ((nbreWorkers < MIN_WORKERS) || (nbreWorkers > MAX_WORKERS))
63         usage(argv[0], "nbre workers incorrect");
64
65     // création du sémaphore
66     sem = my_semget(nbreWorkers);
67
68     printf("Le sémaphore est crée et vous devez lancer %d workers\n",
69            nbreWorkers);
70     printf("Lorsque tous les workers ont terminé, appuyez sur <Entrée>\n");
71
72     printf("—> ");
73     getchar();
74
75     // destruction du sémaphore
76     // si vous êtes curieux, faites-le pendant que les workers l'utilisent
77     my_destroy(&sem);
78
79     printf("Fin du master\n");
80
81     return EXIT_SUCCESS;
82 }

```

**Listing 15.21 – exo10\_master.c**

```

1 // pour des problèmes de compilation
2 #define _XOPEN_SOURCE
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <assert.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 #include <sys/IPC.h>
10 #include <sys/sem.h>
11
12 #include "Semaphore.h"
13 #include "exo10_comm.h"
14
15 const int MIN_SECONDES = 3;
16 const int MAX_SECONDES = 9;
17
18
19 // -----
20 // Il faut récupérer le sémaphore avec une clé générée par ftok et le
21 // le fichier fourni dans exo7_comm.h
22 // Il faut être rigoureux : le sémaphore doit déjà exister sinon
23 // c'est une erreur
24 static Semaphore my_semget()
25 {
26     Semaphore s = sema_creerExistant(MON_FICHIER, PROJ_ID, 1);
27     return s;
28 }
29
30 // -----
31 // Le code doit attendre tous les autres workers
32 static void my_barriere(Semaphore s)
33 {
34     sema_prendre(s, 0);
35     sema_attendre(s, 0);
36 }
37
38
39 // -----
40 int main()
41 {
42     int tempsAttente;
43     Semaphore sem;
44
45     // récupération du sémaphore
46     sem = my_semget();
47
48     // simule la 1re partie du code
49     srand(getpid());
50     tempsAttente = MIN_SECONDES + rand() % (MAX_SECONDES - MIN_SECONDES +
1);
51     printf("1re partie : je bosse pendant %d seconde(s)\n", tempsAttente);
52     printf("                Zzz...\n");
53     sleep(tempsAttente);
54     printf("                j'ai fini.\n");

```

```

55
56     printf("\n");
57     printf("J'attends les autres\n");
58
59 // appel de la fonction implémentant la barrière
60 my_barriere(sem);
61
62     printf("\n");
63     printf("Tout le monde a franchi la barrière\n");
64     printf("\n");
65
66 // simule la 2me partie du code
67 srand(getpid());
68 tempsAttente = 1 + rand() % MIN_SECONDES;
69 printf("2me partie : je bosse pendant %d seconde(s)\n", tempsAttente);
70 printf("                  Zzz...\n");
71 sleep(tempsAttente);
72 printf("                  j'ai fini.\n");
73
74     printf("\n");
75     printf("Fin du worker\n");
76
77 // ici appeler le destructeur de Semaphore sans supprimer le sémaphore
78 IPC
79 sema_detruireSansSupression(&sem);
80
81     return EXIT_SUCCESS;
}

```

**Listing 15.22 – exo10\_worker.c**

# Chapitre 16

## Threads

### Résumé et cadre de travail

Dans ce TP il s'agit de manipuler les processus légers (appelés également threads).

De tels processus ont besoin (généralement) de se synchroniser via des sémaphores ou des mutex. Outre les sémaphores IPC, les threads possèdent leur propre mécanisme de mutex.

Rappel d'une directive importante : pour tout appel système, (ouverture de fichier, lancement d'un thread, ...) vous devez tester la valeur de retour pour détecter toute anomalie ; on ne présuppose jamais qu'un appel système fonctionne. Un assert suffira amplement.

# Principe de fonctionnement des threads

## Point de cours

Un thread est un processus dit léger qui s'exécute au sein d'un processus lourd ; et plusieurs threads peuvent s'exécuter concurremment dans un processus lourd.

La mémoire du processus lourd est partagée par tous les threads.

Techniquement, lorsqu'un thread est lancé, il exécute le code d'une fonction (qui s'exécute donc en concurrence avec les autres threads). Lorsque la fonction se termine, le thread se termine par la même occasion.

Le thread principal est celui qui exécute le programme principal (fonction *main*). Si celui-ci se termine, alors tous les autres threads sont terminés de manière brutale (au contraire des processus lourds avec *fork*). Le thread principal doit donc attendre la fin de tous les autres threads avant de se terminer lui-même (et on retrouve ici un comportement analogue à celui des *fork*, si ce n'est que le devoir d'attente des threads est absolu).

Nous utiliserons obligatoirement les fonctions suivantes :

- *pthread\_create* : pour créer et lancer un thread
- *pthread\_join* : pour attendre la fin d'un thread fils
- *pthread\_exit* : pour terminer (proprement) un thread (un thread se termine également proprement lorsqu'il arrive à la fin de sa fonction support)
- *pthread\_self* : renvoie l'identifiant du thread courant (analogie à *getpid*)

Voici d'autres fonctions que nous n'utiliserons pas :

- *pthread\_cancel* : arrêter un autre thread
- *pthread\_detach* : rendre un thread indépendant de son créateur
- *pthread\_equal* : compare les identifiants de deux threads
- *pthread\_attr\_<something>* : pour paramétriser le comportement d'un thread (nous utiliserons uniquement le paramétrage par défaut)

# Création de threads

## Point de cours

Un thread exécute une fonction dont le prototype est obligatoirement :

```
void * nom_fonction(void *arg);
```

Les arguments de la fonction sont fournis via un pointeur sans type que nous étudierons ultérieurement.

La fonction support d'un thread retourne un pointeur sans type (NULL par convention si on n'attend aucun retour).

La fonction *pthread\_create* prend 4 arguments :

- un pointeur sur une variable de type *pthread\_t* qui sera remplie avec l'identifiant du thread créé (pour être utilisé par *pthread\_join* par exemple).
- le paramétrage du thread : *NULL* pour nous.
- le nom de la fonction support du thread.
- les arguments de la fonction support du thread : *NULL* dans un premier temps pour nous.

La fonction renvoie 0 en cas de succès.

la fonction *pthread\_join* attend la fin d'un thread fils et prend 2 arguments :

- une variable de type *pthread\_t* qui identifie la thread à attendre.
- un paramètre permettant de récupérer le code de retour du thread : *NULL* si ce code ne nous intéresse pas.

La fonction renvoie 0 en cas de succès.

**Question :** Écrivez un programme qui lance plusieurs threads ; chaque thread affiche son pid. Le squelette exo3.c est fourni. Les différents threads affichent-ils un pid différent ? La réponse est non, pourquoi ?

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4
5 //Commun aux fonctions pthread_<...>:
6 #include <pthread.h>
7 //Fonction sleep() et getpid():
8 #include <unistd.h>
9 //Fonction getpid():
10 #include <sys/types.h>
11
12
13 // nombre de threads à lancer
14 #define NB_THREADS 3
15
16 // fonction support d'un thread
17 // éventuellement faites une temporisation d'une seconde avant l'affichage
18 // tous les threads lanceront cette fonction
19 void* codeSupport()
20 {
21     sleep(2);
22     printf("Hello World! I'm thread number %ld, my pid is %i \n",
            pthread_self(), getpid());
```

```

23     sleep(1);
24
25     return NULL;
26 }
27
28 int main()
29 {
30     // On déclare ici un tableau de pthread_t pour stocker tous les
31     // identifiants des threads
32     pthread_t threads_pid[NB_THREADS];
33
34     // lancement des threads
35     // ne pas oublier de tester le code de retour
36     for (int i = 0; i < NB_THREADS; i++)
37     {
38         int ret = pthread_create(&(threads_pid[i]), NULL, codeSupport, NULL);
39         assert(ret == 0);
40     }
41
42     // attente de la fin des threads
43     // (testez le programme sans attente, éventuellement avec une
44     // temporisation dans le code du thread)
45     // ne pas oublier de tester le code de retour
46     for (int i = 0; i < NB_THREADS; i++)
47     {
48         int ret = pthread_join(threads_pid[i], NULL);
49         assert(ret == 0);
50     }
51
52     printf("Fin thread principal\n");
53
54 }
```

**Listing 16.1** – Code source des threads affichant leurs identifiants et pid respectifs

```

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP8/CODE_ETU/EXO_03$ ./ex3
Hello World! I'm thread n°140167255639808, my pid is 46642
Hello World! I'm thread n°140167247247104, my pid is 46642
Hello World! I'm thread n°140167238854400, my pid is 46642
Fin thread principal
```

**FIGURE 16.1** – Résultat du code dans Bash

On remarque que les pid sont les mêmes. En effet, ce résultat qui peut paraître s'explique par le fait qu'un thread s'exécute au sein d'un processus lourd. Il a donc le pid du processus lourd dans lequel il est inclus.

## Création de threads avec paramètres

### Point de cours

Lorsqu'on veut envoyer des paramètres à la fonction thread (ce qui est souvent le cas), il faut les passer via le 4<sup>me</sup> paramètre de la fonction *pthread\_create*.

La fonction thread n'accepte qu'un paramètre qui est un pointeur sur ce qu'on veut (*void \**). S'il n'y a qu'un seul paramètre, il suffit de passer un pointeur dessus. S'il y a plusieurs paramètres à transmettre, il faudra passer par une structure intermédiaire et transmettre un pointeur sur cette structure.

La fonction thread reçoit un *void \**; elle devra caster ce pointeur dans le type attendu.

Attention il n'y a aucune vérification du compilateur (c'est impossible), il faut donc être vigilant pour caster dans le bon type.

Il ne faut lire le libellé d'une question qu'une fois les précédentes programmées (il y a des éléments de réponse d'une question dans les questions suivantes).

Les squelettes des programmes sont fournis.

---

**Question :** Il s'agit de passer à la fonction thread son numéro d'ordre (1 pour le premier lancement, 2 pour le deuxième et ainsi de suite). Pour cela on déclare un compteur dans la fonction main et on passe un pointeur dessus lors de la création des thread). Le thread affiche le numéro qu'elle reçoit (après une attente d'une seconde).

Il se passe un comportement a priori surprenant ; expliquez-le.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include <pthread.h>
5 #include <unistd.h>
6 #include <assert.h>
7
8 // nombre de threads à lancer
9 #define NB_THREADS 3
10
11 // Fonction support d'un thread
12 // Tous les threads lanceront cette fonction
13 // Code
14 //    le paramètre de la fonction est un void *
15 //    il faut le caster en int *, avec un code qui ressemble à
16 //        int *p = (int *) arg;
17 //    on fait une temporisation d'une seconde avant l'affichage
18 //    on affiche la valeur du pointeur (p) et de l'entier pointé (*p)
19
20 void* codeThread(void* args)
21 {
22     sleep(2);
23     int* n = (int*) args;
24     printf("Hello World! I'm thread number %i\n", *n);
25     sleep(1);
```

```

26         return NULL;
27     }
28
29
30 int main()
31 {
32     // On déclare ici un tableau de pthread_t pour stocker tous les
33     // identifiants des threads
34     pthread_t threads[NB_THREADS];
35
36     // le compteur pour passer un numéro aux threads
37     int compteur = 0;
38
39     // lancement des threads
40     for (int i = 0; i < NB_THREADS; i++)
41     {
42         compteur++;
43         // et donc on passe un pointeur sur compteur au thread
44
45         int create_ret = pthread_create(&(threads[i]), NULL, codeThread, &
46         compteur);
47         assert(create_ret == 0);
48     }
49
50     // attente de la fin des threads
51     // (testez le programme sans attente, éventuellement avec une
52     // temporisation dans le code du thread)
53     for (int i = 0; i < NB_THREADS; i++)
54     {
55         int join_ret = pthread_join(threads[i], NULL);
56         assert(join_ret == 0);
57     }
58
59     printf("Fin thread principal\n");
60     return EXIT_SUCCESS;
}

```

**Listing 16.2** – Code source de la tentative d'afficher un compteur illustrant le caractère parallélisé des threads

```

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP8/CODE_ETU/EXO_04$ ./ex4a
Hello World! I'm thread number 3
Hello World! I'm thread number 3
Hello World! I'm thread number 3
Fin thread principal

```

**FIGURE 16.2** – Résultat du code dans Bash

On remarque que chaque thread affiche 3, comme si le compteur avait atteint son maximum avant que les threads ne se réveillent et affichent le compteur. Ce qui est en fait le cas. De plus, comme on passe un pointeur vers compteur, les threads pointent vers la même zone mémoire qui est pointée vers le boucle for. Il y a donc peu de chances qu'on ait le résultat voulu où on voit apparaître 1, 2, 3 (peu importe l'ordre.)

---

**Question :** Pour pallier le problème précédent, une solution est que dans le main il y ait un tableau de numéros : chaque thread recevra un pointeur sur une case qui lui est propre.  
Programmez le nouveau code.

Lancez plusieurs fois le programme pour voir si les affichages sont toujours dans le même ordre.

**Réponse :**

```
1 // un tableau de compteurs pour que chaque thread ait son propre
2 // compteur
3 int compteurs[NB_THREADS];
4
5 // pré-initialisation des compteurs
6 for (int i = 0; i < NB_THREADS; i++)
7     compteurs[i] = i+1;
8
9 // lancement des threads
10 for (int i = 0; i < NB_THREADS; i++)
11 {
12     // et donc on passe un pointeur sur une case différente chaque fois
13     int create_ret = pthread_create(&(threads[i]), NULL, codeThread, &(compteurs[i]));
14     assert(create_ret == 0);
15 }
```

**Listing 16.3** – Extrait du code source affichant le compteur (le reste du code est identique au code cité précédemment)

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP8/CODE_ETU/EXO_04$ ./ex4b
Hello World! My number is 1
Hello World! My number is 2
Hello World! My number is 3
Fin thread principal
```

**FIGURE 16.3** – Résultat du code dans Bash

Ici nous avons eu de la chance nous avons obtenu 1, 2, 3. Mais les threads étant parallèles ils sont en concurrences les uns avec les autres. À certaines exécutions du programmes j'ai donc ainsi obtenu 2, 1, 3 ou d'autres combinaisons...

---

**Question :** Reprenez le programme précédent, mais faites volontairement une erreur de type : on envoie toujours un entier au thread mais ce dernier le caste en float. Regardez le résultat.

Réponse :

```
1 void* codeThread( void* args )
2 {
3     float* n = ( float* ) args;
4     sleep(2);
5     printf("Hello World! My number is %f\n", *n);
6     sleep(1);
7
8     return NULL;
9 }
```

**Listing 16.4** – On a modifié codeThread pour cast les arguments vers un type qui ne lui correspond pas; le reste du code est identique

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP8/CODE_ETU/EXO_04$ ./ex4c
Hello World! My number is 0.000000
Hello World! My number is 0.000000
Hello World! My number is 0.000000
Fin thread principal
```

**FIGURE 16.4** – Résultat du cast malencontreux dans Bash

## Création de threads avec paramètres (bis repetita)

### Question :

Le but est le suivant :

- la fonction main déclare un tableau d'entiers, tableau qui a autant de cases qu'il y a de threads.
- chaque thread a la charge d'une case du tableau.
- chaque thread reçoit 2 entiers, il doit calculer la somme des entiers de l'intervalle et stocker le résultat dans la case du tableau qui lui est attribuée.
- le main affiche le tableau une fois les calculs finis.

Il y a donc plusieurs paramètres à passer à un thread :

- un pointeur sur la case qu'il doit remplir
- le début de l'intervalle
- la fin de l'intervalle

Il faut donc passer par une structure intermédiaire. Le squelette est fourni.

### Réponse :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include <pthread.h>
5 #include <assert.h>
6
7 // nombre de threads à lancer
8 #define NB_THREADS 4
9
10 // structure pour passer les paramètres aux pthread :
11 // . un pointeur sur l'entier devant recevoir le résultat
12 // . le début de l'intervalle
13 // . la fin de l'intervalle
14 typedef struct
15 {
16     int lowBound;
17     int maxBound;
18     int* sum;
19 } ThreadData;
20
21 // Fonction support d'un thread
22 // Tous les threads lanceront cette fonction
23 // Code
24 // le paramètre de la fonction est un void *
25 // il faut le caster en ThreadData *
26 // on calcule la somme des entiers
27 // on stocke le résultat dans la zone mémoire fournie
28
29 void* codeThread(void* args)
30 {
31     ThreadData* data = (ThreadData*) args;
32
33     for(int i = data->lowBound; i <= data->maxBound; i++)
34         *(data->sum) += i;
35
36     return NULL;
```

```

37 }
38
39 int main()
40 {
41     // Débuts et fins des intervalles (attention si on change NB_THREADS)
42     int debuts[NB_THREADS] = {3, 9, -3, 5};
43     int fins[NB_THREADS] = {7, 12, 2, 10};
44     // tableau rempli par les threads
45     int results[NB_THREADS];
46
47     // On déclare ici un tableau de pthread_t pour stocker tous les
48     // identifiants des threads
49     pthread_t threads[NB_THREADS];
50
51     // et un tableau de struct pour que chaque thread ait sa propre
52     // structure
53     ThreadData datas[NB_THREADS];
54
55     // pré-initialisation des données
56     for (int i = 0; i < NB_THREADS; i++)
57     {
58         // il faut initialiser datas[i] avec :
59         // . le début de l'intervalle : debuts[i];
60         // . la fin de l'intervalle : fins[i]
61         // . un pointeur sur l'entier devant recevoir le résultat
62
63         results[i] = 0; //Pour éviter résultat incohérent dans codeThread
64         datas[i].lowBound = debuts[i];
65         datas[i].maxBound = fins[i];
66         datas[i].sum = &(results[i]);
67     }
68
69     // lancement des threads
70     for (int i = 0; i < NB_THREADS; i++)
71     {
72         // et donc on passe un pointeur sur une struct différente chaque
73         // fois
74         int create_ret = pthread_create(&(threads[i]), NULL, codeThread, &
75             datas[i]));
76         assert(create_ret == 0);
77     }
78
79     // attente de la fin des threads
80     // (testez le programme sans attente, éventuellement avec une
81     // temporisation dans le code du thread)
82     for (int i = 0; i < NB_THREADS; i++)
83     {
84         int join_ret = pthread_join(threads[i], NULL);
85         assert(join_ret == 0);
86     }
87
88     // Affichages résultats
89     printf("Fin thread principal : résultats\n");
90     printf("Attendus : [ ");
91     for (int i = 0; i < NB_THREADS; i++)
92     {

```

```

90     int res = 0;
91     for (int v = debuts[i]; v <= fins[i]; v++)
92         res += v;
93     if (i != 0)
94         printf(", ");
95     printf("%3d", res);
96 }
97 printf("]\n");
98 printf("Obtenus : [ ");
99 for (int i = 0; i < NB_THREADS; i++)
100 {
101     if (i != 0)
102         printf(", ");
103     printf("%3d", results[i]);
104 }
105 printf("]\n");
106
107 return EXIT_SUCCESS;
108 }
```

**Listing 16.5** – Code source d'un programme utilisant un thread qui reçoit plusieurs paramètres simultanément par l'intermédiaire d'une structure

# Mutex posix

## Point de cours

Qui dit mémoire partagée dit synchronisation, par exemple :

- pour qu'un processus attende un autre
- pour qu'une zone mémoire ne puisse pas être accédée par plus d'un thread à la fois (exclusion mutuelle)

Il est possible d'utiliser les sémaphores IPC. Mais ces derniers sont destinés à la communication entre les processus lourds et nécessitent beaucoup de ressources système.

Les threads ont une version particulièrement restreinte des sémaphores : les mutex.

Voici les caractéristiques d'un mutex :

- il ne fonctionne qu'au sein d'un seul processus lourd, pour ses threads
- il est moins gourmand en ressources qu'un sémaphore IPC
- il ne peut prendre que deux valeurs : 0 ou 1. On parle plutôt d'états "locked" (verrouillé) et "unlocked" (déverrouillé).
- seul le thread qui a verrouillé le mutex a le droit de le déverrouiller
- un thread voulant verrouiller un mutex déjà verrouillé est bloqué jusqu'à son déverrouillage (fonctionnement normal d'un sémaphore)
- déverrouiller un mutex déjà déverrouillé est une erreur
- bref il sert uniquement aux sections critiques

Un mutex est donc beaucoup moins puissant qu'un sémaphore, mais pour les sections critiques il est préférable de l'utiliser (vu son faible impact sur les ressources).

Nous utiliserons les fonctions suivantes :

- initialisation : par affectation directe lors de la déclaration de la variable de type mutex :

```
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER;
```

- *pthread\_mutex\_lock* : verrouillage (et blocage si déjà verrouillé)
- *pthread\_mutex\_trylock* : comme *lock* mais ne se bloque pas si le mutex est déjà verrouillé, il renvoie un code d'erreur à la place
- *pthread\_mutex\_unlock* : déverrouillage
- *pthread\_mutex\_destroy* : libère les ressources d'un mutex (qui ne doit donc plus être utilisé)

Nous n'utiliserons pas les fonctions du type *pthread\_mutexattr\_<something>*.

Les squelettes des programmes sont fournis.

---

**Question :** Écrivez un programme qui lance plusieurs threads, chacun faisant une grande quantité d'incrémentations (+1) sur une variable partagée.

L'adresse de la variable partagée (qui est déclarée en local dans le main) est passée en paramètre au thread. Il en est de même pour le nombre d'incrémentations à effectuer.

Il n'y a pas de section critique pour protéger l'accès à la variable.

Vérifiez si le résultat est cohérent. A priori ce n'est pas le cas, pourquoi ?

Réponse :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6 #include <pthread.h>
7
8
9 // nombre de threads à lancer
10 #define NB_THREADS 10
11
12 // nombre d'incrémentations par thread
13 #define NB_ITERATIONS 10*1000*1000
14
15 // structure pour passer les paramètres aux pthread :
16 //   . un pointeur sur l'entier devant recevoir le résultat
17 //   . le nombre d'incrémentations
18 typedef struct
19 {
20     int *result;
21     int nbIterations;
22 } ThreadData;
23
24 // Fonction support d'un thread
25 // Tous les threads lanceront cette fonction
26 // Code
27 //   le paramètre de la fonction est un void *
28 //   il faut le caster en ThreadData *
29 void* codeThread(void* arg)
30 {
31     ThreadData *data = (ThreadData*) arg;
32
33     for(int i = 0; i < data->nbIterations; i++)
34     {
35         (*(data->result))++;
36     }
37
38     return NULL;
39 }
40
41 int main()
42 {
43     // variable partagée (ne pas oublier de l'initialiser à 0)
44     int result = 0;
45
46     // On déclare ici un tableau de pthread_t pour stocker tous les
47     // identifiants des threads
48     pthread_t tabId[NB_THREADS];
49     // et un tableau de struct pour que chaque thread ait sa propre
50     // structure
51     ThreadData datas[NB_THREADS];
52
53     // pré-initialisation des données
54     for (int i = 0; i < NB_THREADS; i++)
```

```

54 {
55     // il faut initialiser datas[i] avec :
56     // . un pointeur sur la variable à incrémenter
57     // . le nombre d'itérations à faire
58     datas[i].result = &result;
59     datas[i].nbIterations = NB_ITERATIONS;
60 }
61
62 // lancement des threads
63 for (int i = 0; i < NB_THREADS; i++)
64 {
65     // et donc on passe un pointeur sur une struct différente chaque
66     // fois
67     int ret = pthread_create(&(tabId[i]), NULL, codeThread, &(datas[i]))
68 );
69     assert(ret == 0);
70 }
71
72 // attente de la fin des threads
73 for (int i = 0; i < NB_THREADS; i++)
74 {
75     int ret = pthread_join(tabId[i], NULL);
76     assert(ret == 0);
77 }
78
79 // Affichage résultat
80 printf("Fin thread principal : résultat\n");
81 printf("Attendu : %d\n", NB_THREADS * NB_ITERATIONS);
82 printf("Obtenu : %d\n", result);
83
84 // Vous avez du remarquer que le programme est complet
85 // Mais pourquoi le résultat obtenu est incorrect ?
86
87 return EXIT_SUCCESS;
}

```

**Listing 16.6** – Code source d'un programme lançant des threads modifiant la même zone mémoire sans mutex pour contrôler l'accès à une section critique ou SC

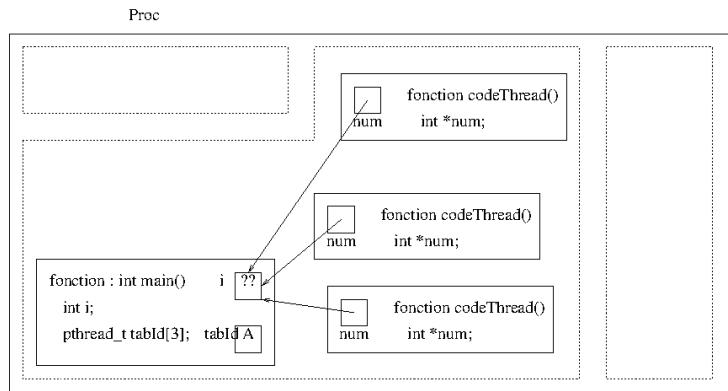
```

crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP8/CODE_ETU/EXO_06$ ./exo6a
Fin thread principal : résultat
Attendu : 100000000
Obtenu : 14357635
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP8/CODE_ETU/EXO_06$ ./exo6a
Fin thread principal : résultat
Attendu : 100000000
Obtenu : 14571052
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP8/CODE_ETU/EXO_06$ ./exo6a
Fin thread principal : résultat
Attendu : 100000000
Obtenu : 16717906
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP8/CODE_ETU/EXO_06$ ./exo6a
Fin thread principal : résultat
Attendu : 100000000
Obtenu : 15989436
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP8/CODE_ETU/EXO_06$ ./exo6a
Fin thread principal : résultat
Attendu : 100000000
Obtenu : 12304383

```

**FIGURE 16.5** – Résultat dans Bash

### Explication :



**FIGURE 16.6** – Figure illustrant la situation

Tous les threads travaillent en parallèle. Sauf que toute écriture ne peut se faire qu'un seul thread à la fois. Ils commencent donc tous par lire 0 puis se disputent l'écriture.

Supposons qu'un thread1 écrivent 1 puis qu'il ait eu le temps d'incrémenter une nouvelle fois et qu'un autre thread2 parvienne finalement à écrire son 1, la valeur de la mémoire partagée est alors finalement repassée à 1.

La valeur de la zone mémoire partagée continue ainsi, certes, à s'incrémenter mais avec de possibles nombreux retours en arrière qui expliquent nos résultats.

---

**Question :** On reprend le programme précédent, mais chaque incrémentation est protégée dans une section critique. Note : le mutex est déclaré en variable globale.

Vérifiez que le résultat est maintenant correct.

Comparez le temps d'exécution avec celui de la première version. Pourquoi est-ce si long ?

**Réponse :**

```
1 // mutex déclaré en global
2 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
3
4 // Fonction support d'un thread
5 // Tous les threads lanceront cette fonction
6 // Code
7 //    le paramètre de la fonction est un void *
8 //    il faut le caster en ThreadData *
9 //    chaque incrémentation est en section critique
10 // exceptionnellement pas de assert pour ne pas ralentir les threads
11 void * codeThread(void * arg)
12 {
13     ThreadData *data = (ThreadData *) arg;
14
15     for (int i = 0; i < data->nbIterations; i++)
16     {
17         pthread_mutex_lock(&mutex);
18         (*(data->result))++;
19         pthread_mutex_unlock(&mutex);
20     }
21
22     return NULL;
23 }
```

**Listing 16.7** – Extrait Code source d'un programme lançant des threads modifiant la même zone mémoire avec mutex pour contrôler l'accès à une SC

On note seulement qu'on détruit le mutex avec l'instruction `pthread_mutex_destroy(&mutex)` après que tous les threads aient terminé pour que le programme se termine proprement.

On remarque que l'exécution du code est beaucoup plus longue parce que chaque thread est bloqué dans son incrémentation tant qu'il n'a pas accès à la SC. De plus, il a 10 000 000 de unlock/lock à faire pour chaque incrémentation ce qui est coûteux en temps.

**Note :** On aurait aussi pu faire les unlock et lock autour de la boucle for et non dedans.

En revanche, les résultats sont maintenant correctes :

```
crex@crex:~/MEGA/Bureau/Etudes/UE3_PAC/TP/TP8/CODE_ETU/EXO_06$ ./exo6b
Fin thread principal : résultat
Attendu : 100000000
Obtenu : 100000000
```

**FIGURE 16.7** – Résultat dans Bash

---

**Question :** Il s'agit du même programme, mais cette fois-ci toutes les incrémentations sont faite sur une variable locale du thread, et la mise à jour de la variable partagée est faite une seule fois (et en section critique) à la fin du thread.

Comparer le nouveau temps d'exécution avec l'ancien.

**Réponse :**

```
1 void * codeThread( void * arg )
2 {
3     ThreadData *data = ( ThreadData * ) arg ;
4     int ajout = 0 ;
5
6     for ( int i = 0; i < data->nbIterations ; i++ )
7         ajout ++ ;
8
9     pthread_mutex_lock(&mutex) ;
10    (*(data->result)) += ajout ;
11    pthread_mutex_unlock(&mutex) ;
12
13    return NULL ;
14 }
```

**Listing 16.8** – Extrait du code source de exo6c.c (n'est pas montré dans cet extrait la destruction du mutex comme mentionné dans la réponse précédente)

Les résultats sont correctes et on constate que cette fois-ci l'exécution est bien plus rapide car il n'y a plus que 10 unlock/lock en tout !

---

**Question :** Il s'agit du même programme, mais cette fois-ci le mutex est déclaré en local dans la fonction main et est passé en paramètre aux threads.

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 // inclure ici les .h nécessaires
4 #include <assert.h>
5 #include <sys/types.h>
6 #include <unistd.h>
7 #include <pthread.h>
8
9
10 // nombre de threads à lancer
11 #define NB_THREADS 10
12
13 // nombre d'incrémentations par thread
14 #define NB_ITERATIONS 10*1000*1000
15
16 // structure pour passer les paramètres aux pthread :
17 //   . un pointeur sur l'entier devant recevoir le résultat
18 //   . le nombre d'incrémentations
```

```

19 // . le sémaphore (un pointeur obligatoirement)
20 typedef struct
21 {
22     int *result;
23     int nbIterations;
24     pthread_mutex_t* mutex;
25 } ThreadData;
26
27
28 // Fonction support d'un thread
29 // Tous les threads lanceront cette fonction
30 // Code
31 // le paramètre de la fonction est un void *
32 // il faut le caster en ThreadData *
33 // les itérations sont faites sur une variable locale
34 // la mise à jour de la variable partagée est en section critique
35 // Faites des assert pour tester les verrouillage et déverrouillage
36 void* codeThread(void* arg)
37 {
38     ThreadData *data = (ThreadData *) arg;
39     int ajout = 0;
40
41     for (int i = 0; i < data->nbIterations; i++)
42         ajout++;
43
44     pthread_mutex_unlock(data->mutex);
45     (*(data->result)) += ajout;
46     pthread_mutex_lock(data->mutex);
47     return NULL;
48 }
49
50 int main()
51 {
52     // mutex déclaré en local
53     pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
54
55     // variable partagée (ne pas oublier de l'initialiser à 0)
56     int result = 0;
57
58     // On déclare ici un tableau de pthread_t pour stocker tous les
59     // identifiants des threads
60     pthread_t tabId[NB_THREADS];
61     // et un tableau de struct pour que chaque thread ait sa propre
62     // structure
63     ThreadData datas[NB_THREADS];
64
65     // pré-initialisation des données
66     for (int i = 0; i < NB_THREADS; i++)
67     {
68         // il faut initialiser datas[i] avec :
69         // . un pointeur sur la variable à incrémenter
70         // . le nombre d'itérations à faire
71         // . le mutex partagé
72         datas[i].result = &result;
73         datas[i].nbIterations = NB_ITERATIONS;
74         datas[i].mutex = &mutex;

```

```

74 }
75
76 // lancement des threads
77 for (int i = 0; i < NB_THREADS; i++)
78 {
79     // et donc on passe un pointeur sur une struct différente chaque
80     // fois
81     int ret = pthread_create(&(tabId[i]), NULL, codeThread, &(datas[i]))
82 );
83     assert(ret == 0);
84 }
85
86 // attente de la fin des threads
87 for (int i = 0; i < NB_THREADS; i++)
88 {
89     int ret = pthread_join(tabId[i], NULL);
90     assert(ret == 0);
91 }
92
93 // pour être propre, on détruit le mutex
94 pthread_mutex_destroy(&mutex);
95
96 // Affichage résultat
97 printf("Fin thread principal : résultat\n");
98 printf("Attendu : %d\n", NB_THREADS * NB_ITERATIONS);
99 printf("Obtenu : %d\n", result);
100
101 return EXIT_SUCCESS;
}

```

**Listing 16.9** – Code source de exo6d.c, le mutex est maintenant local au main mais il doit être ajouté dans la structure car il devient un paramètre des threads lancé par le main

---

## Sémaphore anonyme (facultatif )

### Point de cours

Il existe également un type sémaphore dédié plus particulièrement aux threads : *sem\_t* .  
Il est aussi possible de partager de tels sémaphores entre des processus lourds mais nous n'utiliserons pas cette possibilité.

Le comportement est très proche de celui des sémaphores IPC, mais d'une puissance moindre cependant :

- la variation de la valeur d'un sémaphore ne se fait que par incrément de  $\pm 1$
- la possibilité de se bloquer jusqu'à ce que la valeur du sémaphore atteigne 0 (zéro) n'existe pas.

Nous utiliserons les fonctions suivantes :

- *sem\_init* : création et initialisation d'un sémaphore
- *sem\_wait* : tentative de décrémenter (-1) un sémaphore
- *sem\_post* : incrémentation (+1) d'un sémaphore
- *sem\_destroy* : libère les ressources d'un sémaphore

Nous n'utiliserons pas les fonctions suivantes :

*sem\_open sem\_close sem\_unlink sem\_trywait sem\_timedwait*

La fonction *sem\_init* prend 3 arguments :

- un pointeur sur une variable de type *sem\_t* qui sera remplie avec l'identifiant du sémaphore créé.
- un paramètre indiquant si le sémaphore est partagé entre des processus lourds : 0 pour nous, indiquant que le sémaphore n'est partagé que par les threads du processus.
- un entier indiquant la valeur initiale.

La fonction renvoie 0 en cas de succès.

**Question :** Le thread principal lance plusieurs threads fils. Un thread fils commence par un temps d'initialisation, puis continue par un temps de travail.

Le thread principal, une fois qu'il a lancé les fils, doit attendre, avant de continuer, que tous les fils aient fini leurs initialisations.

Le squelette est fourni : il ne reste qu'à gérer le sémaphore et l'attente du thread principal

**Réponse :**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include <assert.h>
5 #include <sys/types.h>
6 #include <unistd.h>
7 #include <pthread.h>
8 #include <semaphore.h>
9
10
11 // nombre de threads à lancer
12 #define NB_THREADS 3
13
14 // structure pour passer les paramètres aux pthread :
```

```

15 // . le numéro du thread
16 // . le sémaphore (un pointeur obligatoirement)
17 typedef struct
18 {
19     int numero;
20     sem_t* sem;
21 } ThreadData;
22
23
24 // Fonction support d'un thread
25 // Tous les threads lanceront cette fonction
26 // Code
27 // le paramètre de la fonction est un void *
28 // il faut le caster en ThreadData *
29 // Faites des assert pour tester les manipulations du sémaphore
30 void * codeThread(void * arg)
31 {
32     ThreadData *data = (ThreadData *) arg;
33
34     // simulation d'un temps d'initialisation
35     sleep(rand()%3 + 1);
36
37     printf("      [%d] commence à travailler\n", data->numero);
38
39     // on prévient le thread principal qu'on commence à travailler
40     int ret = sem_post(data->sem);
41     assert(ret != -1);
42
43
44     sleep(rand()%3 + 1);
45     printf("      [%d] a fini de travailler\n", data->numero);
46
47     return NULL;
48 }
49
50 int main()
51 {
52     // sémaphore déclaré en local
53     sem_t sem;
54     int init_ret = sem_init(&sem, 0, 0);
55     assert(init_ret != -1);
56
57     // On déclare ici un tableau de pthread_t pour stocker tous les
58     // identifiants des threads
59     pthread_t tabId[NB_THREADS];
60     // et un tableau de struct pour que chaque thread ait sa propre
61     // structure
62     ThreadData datas[NB_THREADS];
63
64     // initialisation génération nombres aléatoires
65     srand(getpid());
66
67     // pré-initialisation des données
68     for (int i = 0; i < NB_THREADS; i++)
69     {
69         // il faut initialiser datas[i] avec :

```

```

70 // . le numéro du thread
71 // . le sémaphore partagé
72 datas[ i ].numero = i+1;
73 datas[ i ].sem = &sem;
74 }
75
76 // lancement des threads
77 for ( int i = 0; i < NB_THREADS; i++)
78 {
79     // et donc on passe un pointeur sur une struct différente chaque
80     // fois
81     int ret = pthread_create(&(tabId[ i ]), NULL, codeThread, &(datas[ i ]));
82     assert( ret == 0 );
83 }
84
85 // attendre que les threads aient fini leurs initialisations
86 printf("J'attends les débuts des threads\n");
87 for ( int i = 0; i < NB_THREADS; i++)
88 {
89     // Attente de chaque fils:
90     int ret = sem_wait(&sem);
91     assert( ret != -1 );
92     printf(" un thread de prêt ( i = %d)\n", i );
93 }
94 printf("Fin attente\n");
95
96 // attente de la fin des threads
97 for ( int i = 0; i < NB_THREADS; i++)
98 {
99     int ret = pthread_join( tabId[ i ], NULL );
100    assert( ret == 0 );
101 }
102
103 // pour être propre, on détruit le sémaphore
104 int ret = sem_destroy(&sem);
105 assert( ret != -1 );
106
107 printf("Fin thread principal\n");
108
109 return EXIT_SUCCESS;
}

```

**Listing 16.10** – Code source de exo7.c où un thread principal (le main) attend que tous ses threads lancés aient commencé à travailler avant de lui-même poursuivre son exécution