Loric Informatique

C

Version 1.1 (créé le 10/12/2022, modifié le 14/01/2024)

C est un langage de programmation impératif, généraliste et de bas niveau. C offre au développeur une marge de contrôle importante sur la machine (notamment sur la gestion de la mémoire) et est de ce fait utilisé pour réaliser les « fondations » (compilateurs, interpréteurs…) des langages plus modernes.

Outils nécessaires :

- Un logiciel de codage (Visual Studio Code ou Notepad++)
- Un compilateur de programmes (GNU GCC Compiler ou MySys)
- Ou un logiciel tout en un (CodeBlocks (fortement recommandé))

Table des matières :

I. Bases
- **I.1. Syntaxe**
- **I.2. Enregistrer et exécuter un programme fichier.c**
- **I.3. Types de variables**
  - **I.3.1. Caractères**
  - **I.3.2. Numériques**
- **I.4. Commentaires**
- **I.5. Opérations**
- **I.6. Les variables**
- **I.7. Les tableaux**
- **I.8. Conditions**
  - **I.8.1. Opérateurs de comparaison**
  - **I.8.2. Tests de conditions**
- **I.9. Boucles**
- **I.10. Les différentes valeurs de contrôles (dans printf et scanf notamment)**

II. Les fonctions, les préprocesseurs et les structures
- **II.1. Les fonctions**
  - **II.1.1. Créer une fonction retournant une valeur du type « type0 »**
  - **II.1.2. Retourner une valeur de la fonction**
  - **II.1.3. Créer une fonction retournant aucune valeur**
  - **II.1.4. Faire un appel à la fonction**
  - **II.1.5. Modifier directement des variables extérieurs grâce à des adresses (pointeurs)**
- **II.2. Les préprocesseurs (ou directives)**
  - **II.2.1. Créer une constante**
  - **II.2.2. Constantes prédéfinies**

II.3. Les structures
II.3.1. Manière simple
II.3.2. Manière rapide
II.4. Les énumérations
II.4.1. Créer une énumération
II.4.2. Affecter une valeur de l'énumération
III. Les instructions
III.1. Instructions de bases
III.2. Les fichiers
IV. Les bibliothèques
IV.0. Importer une bibliothèque (fichier c + fichier h)
IV.1. stdlib (#include ‹stdlib.h›)
IV.2. string (#include ‹string.h›)
IV.3. assert (#include ‹assert.h›)
IV.4. math (#include ‹math.h›)
IV.5. pthread (#include ‹pthread.h›)
IV.5.1. Syntaxe de base
IV.5.2. Commandes
IV.5.3. Les sections critiques (Mutex)
IV.6. unistd (#include ‹unistd.h›)
IV.6.1. Syntaxe de base
IV.6.2. Commandes
I.7. Sémaphores (#include ‹semaphore.h›)

I. Bases

I.1. Syntaxe

```
#include ‹stdio.h›

int main(void){
    instruction1;
    instruction2;
    instruction3;
    return 0;
}
```

I.2. Enregistrer et exécuter un programme *fichier.c*

Dans la console, tapez la commande suivante :

`gcc fichier.c -o fichier`

I.3. Types de variables

I.3.1. Caractères

Type	Minimum	Maximum	Description
`signed char[i]` (texte)	0	*i*	Chaîne de caractères entre ""

`signed char` (1o)	-128	127	Nombre entier ou caractère (1 seul) entre ''
`unsigned char` (1o)	0	256	Nombre entier ou caractère (1 seul) entre ''

I.3.2. Numériques

Type	Minimum	Maximum	Description
`int` (2o (32-bit))	-32 768	32 767	Nombre entier
`int` (4o (64-bit))	-2 147 483 648	2 147 483 647	Nombre entier
`unsigned int` (2o (32-bit))	0	65 535	Nombre entier
`unsigned int` (4o (64-bit))	0	4 294 967 295	Nombre entier
`short` (2o)	-32 768	32 767	Nombre entier
`unsigned short` (2o)	0	65 535	Nombre entier
`long` (4o)	-2 147 483 648	2 147 483 647	Nombre entier
`unsigned long` (4o)	0	4 294 967 295	Nombre entier
`long long` (8o)	−9 223 372 036 854 775 808	9 223 372 036 854 775 807	Nombre entier
`unsigned long long` (8o)	0	18 446 744 073 709 551 615	Nombre entier
`float` (4o)	$-3.4_{x10}{}^{38}$f	$3.4_{x10}{}^{38}$f	Nombre décimal
`double` (8o) (2 fois plus précis que float)	$-1.7_{x10}{}^{308}$	$1.1_{x10}{}^{308}$	Nombre décimal
`long double` (10o)	$-1.1_{x10}{}^{4932}$	$1.1_{x10}{}^{4932}$	Nombre décimal

I.4. Commentaires

```
//Commentaire tenant sur une ligne (en -C99)

/*
```

```
Commentaire pouvant être sur une ou plusieurs lignes
*/
```

I.5. Opérations

Instruction	Description
`1 + 2`	Renvoie 3
`3 - 1`	Renvoie 2
`6 * 4`	Renvoie 24
`5.0 / 2.0`	Renoie 2.5
`5 / 2`	Renvoie 2 (le quotient sans décimal)
`5 % 2`	Renvoie 1 (le reste de la division)

I.6. Les variables

Instruction	Description
type variable = *valeur*;	Déclare une nouvelle variable (à mettre au début d'une fonction en -ainsi)
int *a*;	Déclare la variable *a* comme int
a = 5;	Affecte 5 à une variable
b = *a*++;	Équivalent à *b* = *a*; *a* = *a* + 1;
b = ++*a*;	Équivalent à *a* = *a* + 1; *b* = *a*;
b = *a*--;	Équivalent à *b* = *a*; *a* = *a* - 1;
b = --*a*;	Équivalent à *a* = *a* - 1; *b* = *a*;
int *a* = (int)*b*;	Convertit le nombre décimal *b* en nombre entier *a*
const float *pi* = 3.14;	Crée une variable constante (non modifiable)
register int *n* = 5;	Insère dans le registre
volatile int *n* = 5;	Insère dans la mémoire RAM

`static type variable = valeur;`	Déclare une nouvelle variable qui n'est jamais détruite (si déjà exécuté, cette instruction est ignorée)
`char caractere = 'c';`	Déclare une variable contenant un caractère
`#define VARIABLE valeur`	À mettre avant la fonction main(), permet de créer une variable globale et constante

I.7. Les tableaux

Instruction	Description
`type tableau[i];`	Crée un tableau vide de *i* éléments (*i* doit être une variable globale en norme -ainsi, définie avec #define)
`int tableau[i] = {0};`	Crée un tableau de *i* éléments (*i* est facultatif) égal à 0 en -c99 ou + (*i* doit être une variable globale en norme -ainsi, définie avec #define)
`char texte[i] = "message";` ou char *texte*[*i*] = {'*m*', '*e*', '*s*', '*s*', '*a*', '*g*', '*e*'};	Déclare une variable de longueur *i* maximum (non obligatoire, mais conseillé) contenant une chaîne de caractère (ne peut pas être modifié selon les versions de C) (Une chaîne de caractère est également un tableau de caractères)

`tableau[i]`
ou
`*(tableau + i)`
	Renvoie la valeur du tableau à la position *i* (l'indice de la première valeur est 0)

`tableau[i] = {[j] = 1};`	Crée un tableau de *i* éléments avec la valeur 1 dans l'indice *j* (*i* et *j* doivent être des variables globales en norme -ainsi, définies avec #define)
`int tableau[i] = {val1, val2…};`	Crée un tableau de i éléments avec des valeurs personnalisées (*i* doit être une variable globale en norme -ainsi, définie avec #define)
`*tableau`	Affiche la première valeur du tableau. Également obligatoire si le tableau est un argument d'une fonction
`sizeof(tableau)`	Renvoie la taille du tableau en octet

Remarque : Un tableau est marqué à la fin par le caractère '\0' dans la mémoire RAM pour indiquer la fin d'un tableau. Tout débordement du tableau pourrait donner accès à une variable aléatoire dans la RAM.

I.8. Conditions

Les conditions renvoient 1 si elle est respectée et 0 sinon

I.8.1. Opérateurs de comparaison

Condition	Description de ce que vérifie la condition
a == *b*	*a* égal à *b*
a < *b*	*a* strictement inférieur à *b*
a > *b*	*a* strictement supérieur à *b*
a <= *b*	*a* inférieur ou égal à *b*
a >= *b*	*a* supérieur ou égal à *b*
a != *b*	*a* n'est pas égal à *b*
||	À mettre entre deux conditions, permet d'avoir une des deux conditions qui doit être vraie
&&	À mettre entre deux conditions, permet d'avoir deux conditions qui doivent être vraie
!*condition*	La condition doit être fausse

I.8.2. Tests de conditions

Instruction	Description

if(*condition1*){
  instruction1;
}
	Si *condition1* est vraie, alors on exécute *instruction1*

if(*condition1*){
  instruction1;
}
else{
  instruction2;
}
	Si *condition1* est vraie, alors on exécute *instruction1*, sinon, on exécute *instruction2*

if(*condition1*){
  instruction1;
}
	Si *condition1* est vraie, alors on exécute *instruction1*, sinon, si *condition2* est vraie, on

`else if(condition2){`
`  instruction2;`
`}`
`else{`
`  instruction3;`
`}`
	exécute *instruction2*, sinon, on exécute *instruction3*

`switch(nombre){`
`  case a:`
`  instruction1;`
`  break;`
`  case b:`
`  case c:`
`  instruction2;`
`  break;`
`  default:`
`  instruction3;`
`}`
	Si *nombre == a*, alors on exécute *instruction1*, sinon, si *nombre == b* ou *c*, on exécute *instruction2*, sinon, on exécute *instruction3*

`a = (condition1) ? 1 : 0;`	Si *condition1* est vraie, *a* prend la valeur 1, sinon 0.

I.9. Boucles

Instruction	Description

`for(int i = d; i < f; i++){`
`  instruction1;`
`}`
	On répète *f-d* fois *instruction1* pour *i* allant de *d* compris à *f* non compris (attention, la déclaration de *i* doit se faire au début pour la norme -ainsi)

`for(int i = d; i < f; i+=p){`
`  instruction1;`
`}`
	On répète (*f-d*)/*p* fois *instruction1* pour i allant de *d* compris à *f* non compris avec pour pas égal à *p*

`while(condition){`
`  instruction1;`
`}`
	On répète jusqu'à ce que *condition* soit fausse (peut ne pas être répété)

`do{`
`  instruction1;`
`}`
`while(condition);`
	On répète jusqu'à ce que *condition* soit fausse (est forcément répété une fois)

`break;`	Permet de sortir d'une boucle sans la terminer (fortement déconseillé)
`continue;`	Permet de revenir au début de la boucle

I.10. Les différentes valeurs de contrôles (dans printf et scanf notamment)

Contrôles (`%controle`)	Type renseigné
`%d ou %i`	int
`%x`d	int avec *x* chiffres maximum
`%u`	unsigned int
`%o`	unsigned int (octal)
`%f`	float (ou double)
`%x`f	float avec *x* chiffres entiers maximum
`%.y`f	float avec *y* chiffres après la virgule maximum
`%x.y`f	float avec *x* chiffres dans la partie entière et *y* chiffres après la virgule maximum
`%lf`	double
`%c`	char
`%s`	char[*i*] (string)
`%x`s	char[*i*] (string de *x* caractères maximum)
`%p`	pointeur
`%b`	Affichage en binaire
`%x`	Affichage en hexadécimal (minuscule)
`%X`	Affichage en hexadécimal (majuscule)

II. Les fonctions, les préprocesseurs et les structures

II.1. Les fonctions

Les fonctions doivent être écrites juste après `#include ‹stdio.h›` ou dans un autre fichier .c accompagné de son fichier .h (voir "Les bibliothèques")

II.1.1. Créer une fonction retournant une valeur du type « *type0* »

```
type0 maFonction(type1 variable1, type2 variable2…){
    instructions;
}
```

II.1.2. Retourner une valeur de la fonction

```
return variable;
```

II.1.3. Créer une fonction retournant aucune valeur

```
void maFonction(type1 variable1, type2 variable2…){
    instructions;
}
```

II.1.4. Faire un appel à la fonction

```
variable = maFonction(valeur1, valeur2…);
```

ou (s'il n'y a pas de variable de retour)

```
maFonction(valeur1, valeur2…);
```

II.1.5. Modifier directement des variables extérieurs grâce à des adresses (pointeurs)

```
void maFonction(type *variable){
    *variable = valeur;
}
```

Dans int main(void) :

```
type variable;
maFonction(&variable);
```

Remarque : il est possible de faire une surcharge de fonctions, c'est-à-dire qu'il est possible de créer deux fonctions identiques avec des paramètres de types différents, ce qui permet au compilateur de choisir la fonction correspondant au type de variables saisies.

II.2. Les préprocesseurs (ou directives)

II.2.1. Créer une constante

```
#define variable __valeur__
```

ou

```
#define VARIABLE valeur
```

II.2.2. Constantes prédéfinies

Constante	Description
__FILE__	Nom du fichier
__LINE__	Ligne du fichier

`__DATE__`	Date de compilation
`__TIME__`	Heure de compilation

II.3. Les structures

Deux manières d'écrire des structures, les deux doivent être écrite après #include ‹stdio.h› ou dans un autre fichier .c accompagné de son fichier .h (voir "Les bibliothèques")

II.3.1. Manière simple

```
struct MaStructure{
    type1 variable1;
    type2 variable2;
    type3 variable3;
}
```

Pour créer une structure :

```
struct MaStructure variable = {valeur1, valeur2, valeur3};
```

II.3.2. Manière rapide

```
typedef struct _MaStructure{
    type1 variable1;
    type2 variable2;
    type3 variable3;
} MaStructure;
```

ou

```
struct _MaStructure{
    type1 variable1;
    type2 variable2;
    type3 variable3;
}
typedef struct _MaStructure MaStructure;
```

Pour créer une structure :

```
MaStructure variable = {valeur1, valeur2, valeur3};
```

Afficher une valeur de variable : `variable.variable1`

Dans une fonction : `(*variable).valeur1` ou `variable->valeur1`

II.4. Les énumérations

II.4.1. Créer une énumération

Exemple :

```
typedef enum JoursSemaine{
    LUNDI,
    MARDI,
    MERCREDI,
    JEUDI,
    VENDREDI,
    SAMEDI,
    DIMANCHE
} JourSemaine;
```

II.4.2. Affecter une valeur de l'énumération

Exemple :

```
JourSemaine jour = MERCREDI;
```

III. Les instructions

III.1. Instructions de bases

Instruction	Description
`printf("texte");`	Affiche un texte dans la console
`printf("texte\n");`	Affiche un texte dans la console avec un retour à la ligne (pour une tabulation, entrez \t)
`printf("%controle", variable);` `printf(("Valeur : %controle", variable);`	Affiche une variable (ici *variable*) dans la console
`scanf("%controle", &variable);`	Demande une valeur avec le retour dans *variable* (non sécurisé)
`etiquette:`	Indiquer un emplacement du programme
`goto etiquette;`	Aller dans un emplacement du programme
`sizeof(variable);`	Renvoyer la taille en octets d'une variable (utile pour l'allocation dynamique avec malloc)
`sprintf(chaine, "Valeur : %controle", variable);`	Écrit du texte dans une chaîne de caractères

III.2. Les fichiers

Instruction	Description

`FILE *fichier = fopen(nom_du_fichier, "mode");`	Ouvre un fichier selon le mode choisi (r : Ouvre en lecture; w : Crée un nouveau fichier et l'ouvre en mode écriture (écrase l'ancien fichier du même nom); a : Ouvre un fichier en mode écriture (en écrivant à la fin du fichier); rb ou wb : Ouvre en binaire)
`fprintf(fichier, "%controle", variable);`	Ecrit le contenu d'une variable (ici *variable*) dans le fichier (en mode écriture)
`fscanf(fichier, "%controle", &variable);`	Lit une ligne du fichier avec le retour dans *variable* (en mode lecture)
`feod(fichier);`	Détermine quand on atteindra la fin du fichier
`variable = fgetc(fichier);`	Lit un caractère
`fgets(variable, longueur, fichier);`	Lit une ligne de longueur maximum *longueur*
`fputc(caractere, fichier);`	Écrit un caractère
`fputs(texte, fichier);`	Écrit une chaîne de caractères
`fclose(fichier);`	Important pour fermer un fichier

IV. Les bibliothèques

IV.0. Importer une bibliothèque (fichier c + fichier h)

Dans votre dossier, créer un fichier .h où vous mettrez d'abord :

```
#ifndef FICHIER_H
#define FICHIER_H
```

avec FICHIER le nom de votre fichier *fichier.h* en majuscule.

Puis la liste des fonctions sous forme :

type myfonction(*type variables*);

(il est également possible d'écrire ses structures directement à l'intérieur de ce fichier .h)

Enfin : `#endif`

(ou écrivez simplement `#pragma once` dans le fichier .h pour laisser faire le compilateur et écrivez à la suite les fonctions)

Dans le même dossier, créer un fichier .c du même nom que le fichier .h, ajouter au début : `#include "`*fichier*`.h"` (avec `#include ‹stdio.h›` et les autres bibliothèques)

et entrez vos fonctions à l'intérieur de ce fichier.

Enfin, entrez : #include "*fichier*.h" dans le fichier principal juste après `#include ‹stdio.h›`

IV.1. stdlib (`#include ‹stdlib.h›`)

stdlib permet d'avoir d'autres fonctionnalités utiles pour le langage C. Il est souvent ajouté au début du programme avec stdio.h car il s'agit d'une des bibliothèques qui est la plus utilisée en C.

Instruction	Utilité
`system("PAUSE");`	Mettre en pause le programme pour lire certaines données dans la console
`rand();`	Renvoyer un nombre aléatoire entre 0 et la constante RAND_MAX (la plus grande valeur que la fonction peut renvoyer sur un système donné)
`srand(time(NULL));`	Réinitialiser les valeurs aléatoires (à utiliser 1 fois dans le programme avant rand(), nécessite également la bibliothèque ‹time.h›)
type `*`*pointeur* `= NULL;` *pointeur* `= (`*type* `*)malloc(`*taille_totale*`);` ou *type* `*`*pointeur* `= (`*type* `*)malloc(`*taille_totale*`);`	Allouer dynamiquement une zone mémoire en octets (taille pouvant être donnée avec sizeof) sur un pointeur (renvoie l'adresse de la zone allouée ou NULL si la fonction échoue)

`free(pointeur);`	Important pour libérer la mémoire
`type *pointeur = (type *)calloc(nombre_cases, taille_case);`	Allouer dynamiquement une zone mémoire en octets pour chaque case du pointeur et les initialise à 0 (renvoie l'adresse de la zone allouée ou NULL si la fonction échoue)
`pointeur = realloc(pointeur, taille_totale);`	Réallouer une zone mémoire (renvoie l'adresse de la zone allouée ou NULL si la fonction échoue)
`exit(EXIT_FAILURE);`	Terminer le programme en libérant les ressources utilisées par le programme et en signalant l'échec du déroulement du programme
`exit(EXIT_SUCCESS);`	Terminer le programme en libérant les ressources utilisées par le programme (est souvent remplacé par return EXIT_SUCCESS;)

IV.2. string (`#include ‹string.h›`)

string permet de faire diverses manipulations avec des chaînes de caractères facilement, sans avoir à créer des fonctions de manipulation. Il peut ne pas être utilisé afin de faire par nous-même les fonctionnalités de la bibliothèque string.

Instruction	Utilité
`memccpy(destination, source, caractere, taille);`	Copier un bloc de mémoire dans un second bloc en s'arrêtant après la première occurrence d'un caractère ou à la longueur saisie
`memchr(memoryBlock, caractere, taille);`	Rechercher la première occurrence d'une valeur dans un bloc de mémoire et renvoie son pointeur
`memcmp(pointeur1, pointeur2, taille);`	Comparer le contenu de deux blocs de mémoire
`memcpy(destination, source, taille);`	Copier un bloc de mémoire dans un second bloc (-ainsi)
`memcpy(restrict destination, restrict source, taille);`	Copier un bloc de mémoire dans un second bloc (-c99)
`memmove(destination, source, taille);`	Copier un bloc de mémoire dans un second (fonctionne même si les deux blocs se chevauchent)

`memset(pointeur, valeur, octets);`	Remplir une zone mémoire, identifiée par son adresse et sa taille, avec une valeur précise
`strcat(mot1, mot2);`	Ajouter une chaîne de caractères à la suite d'une autre chaîne
`strchr(chaine, caractere);`	Rechercher la première occurrence d'un caractère dans une chaîne de caractères.
`strcmp(mot1, mot2);`	Comparer deux chaînes de caractères et de savoir si la première est inférieure, égale ou supérieure à la seconde
`strcoll(mot1, mot2);`	Comparer deux chaînes en tenant compte de la localisation en cours
`strcpy(variable, chaine);` `strcpy(destination, source);`	Copier une chaîne de caractères
`strcspn(chaine, caractere);`	Renvoyer la longueur de la plus grande sous-chaîne (en partant du début de la chaîne initiale) ne contenant aucun des caractères spécifiés dans la liste des caractères en rejet
`strdup(chaine);`	Dupliquer la chaîne de caractères passée en paramètre
`strlen(chaine);`	Calculer la longueur de la chaîne de caractères
`strncat(destination, source, taille);`	Ajouter une chaîne de caractères à la suite d'une autre chaîne en limitant le nombre maximum de caractères copiés (-ainsi)
`strncat(restrict destination, restrict source, taille);`	Ajouter une chaîne de caractères à la suite d'une autre chaîne en limitant le nombre maximum de caractères copiés (-c99)
`strncmp(chaine1, chaine2, taille);`	Comparer deux chaînes de caractères dans la limite de la taille spécifiée en paramètre
`strncpy(destination, source, taille);`	Copier, au maximum, les *n* premiers caractères d'une chaîne de caractère dans une autre (-ainsi)
`strncpy(restrict destination, restrict source,taille);`	Copier, au maximum, les *n* premiers caractères d'une chaîne de caractère dans une autre (-c99)
`strndup(source, n);`	Dupliquer au maximum *n* caractères de la chaîne passée en paramètre

`strpbrk(chaine, caractere);`	Rechercher dans une chaîne de caractères la première occurrence d'un caractère parmi une liste de caractères autorisés
`strrchr(source, caractere);`	Rechercher la dernière occurrence d'un caractère dans une chaîne de caractères
`strspn(chaine, listecaracteres);`	Renvoyer la longueur de la plus grande sous-chaîne (en partant du début de la chaîne initiale) ne contenant que des caractères spécifiés dans la liste des caractères acceptés
`strstr(source, mot);`	Rechercher la première occurrence d'une sous-chaîne dans une chaîne de caractères principale
`strtok(chaine, separateur);`	Extraire, un à un, tous les éléments syntaxiques (les tokens) d'une chaîne de caractères (-ainsi)
`strtok(restrict chaine, restrict separateur);`	Extraire, un à un, tous les éléments syntaxiques (les tokens) d'une chaîne de caractères (-c99)
`strxfrm(destination, source, taille);`	Transformer les *n* premiers caractères de la chaîne source en tenant compte de la localisation en cours et les place dans la chaîne de destination (-ainsi)
`strxfrm(restrict destination, restrict source, taille);`	Transformer les *n* premiers caractères de la chaîne source en tenant compte de la localisation en cours et les place dans la chaîne de destination (-c99)

IV.3. assert (`#include ‹assert.h›`)

assert permet de vérifier le fonctionnement d'un algorithme en faisant des tests de conditions.

Instruction	Utilité
`assert(condition);`	Vérifier que *condition* est vraie, sinon, retourne une erreur

IV.4. math (`#include ‹math.h›`)

math permet d'utiliser les fonctions de base de mathématiques.

Instruction	Utilité
`pi`	Obtenir la valeur de π

sqrt(*nombre*)	Utiliser la racine carrée
log(*nombre*)	Utiliser la fonction logarithme
exp(*nombre*)	Utiliser la fonction exponentielle
cos(*radians*)	Utiliser la fonction cosinus
sin(*radians*)	Utiliser la fonction sinus
tan(*radians*)	Utiliser la fonction tangente
acos(*nombre*)	Utiliser la fonction cosinus^{-1}
asin(*nombre*)	Utiliser la fonction sinus^{-1}
atan(*nombre*)	Utiliser la fonction tangente^{-1}

IV.5. pthread (`#include ‹pthread.h›`)

pthread permet de créer des threads pouvant s'exécuter en parallèle avec le programme principal, tout en ayant une mémoire partagée.

> Attention : Ne fonctionne que sous Windows

IV.5.1. Syntaxe de base

```
#include ‹stdio.h›
#include ‹pthread.h›

pthread_t tableauDeThreads[nbThreads];

void* maFonction(void* parametre){
    int* intParametre = (int*)parametre;
    int variable = *intParametre;
    instructions;
    pthread_exit(0);
    return NULL;
}

int main(){
    int parametres[nbThreads];
    for(int i = 0; i ‹ nbThreads; i++){
        parametres[i] = i;
    }
    for(int i = 0; i ‹ nbThreads; i++){
        pthread_create(&tableauDeThreads[i], NULL, &maFonction, &parametres[i]);
    }
    for(int i = 0; i ‹ nbThreads; i++){
        pthread_join(tableauDeThreads[i], NULL);
    }
    instructions;
```

```
    return 0;
}
```

IV.5.2. Commandes

Instruction	Utilité
`pthread_t tableauDeThreads[nbThreads];`	Déclarer une variable qui va contenir tous les threads
`pthread_create(&tableauDeThreads[i], NULL, &maFonction, &parametres[i]);`	Créer un thread
`pthread_exit(0);`	Terminer proprement un thread
`pthread_join(tableauDeThreads[i], NULL);`	Attendre qu'un thread se termine

IV.5.3. Les sections critiques (Mutex)

Créer une section critique permet de modifier une variable sans qu'un autre thread n'essaie de modifier la même variable en même temps, sous risque de perdre des informations. Le mutex va donc mettre provisoirement en pause les autres threads pendant la réalisation des instructions.

Instruction	Utilité
`pthread_mutex_t monMutex; pthread_mutex_init(&monMutex, NULL);`	Déclarer un mutex et l'initialiser
`pthread_mutex_lock(&monMutex);`	Entrer en section critique
`pthread_mutex_unlock(&monMutex);`	Sortir de la section critique

IV.6. unistd (`#include ‹unistd.h›`)

unistd permet de créer des processus pouvant s'exécuter en parallèle avec le programme principal, sans avoir de mémoire partagée (chaque processus créés est le fils du processus appelant la fonction fork, il n'a donc pas accès aux autres variables et doit sauvegarder les variables dans un fichier afin d'être accessible par le processus père).

> Attention : Ne fonctionne que sous Linux

IV.6.1. Syntaxe de base

```
#include ‹stdio.h›
#include ‹unistd.h›

int main(){
    int tableauDePids[nbProcessus];
    int id = -1;
```

```
    int pid;
    int i = 0;
    while(id == -1 && i ‹ nbProcessus){
        pid = fork();
        if(pid != 0){
            tableauDePids[i] = pid;
            i = i + 1;
        }
        else{
            id = i;
        }
    }
    if(id == -1){
        instructionsDuPere;
        for(int i = 0; i ‹ nbProcessus; i++){
            waitpid(-1, NULL, 0);
        }
        instructions;
    }
    else{
        instructionsDuFils;
    }
    return 0;
}
```

IV.6.2. Commandes

Instruction	Utilité
`pid = fork();`	Créer un processus et renvoie le pid du nouveau processus pour le processus qui l'a créé et 0 pour le nouveau processus
`getpid();`	Permet d'avoir le pid du processus qui l'exécute
`int pidPere = getpid();`	À mettre au début de la fonction main, permet d'avoir le pid du père facilement
`int pidFils = getpid();`	À mettre dans les instructions des fils, permet d'avoir le pid actuel du fils
`pidTermine = waitpid(-1, NULL, 0);`	Permet d'attendre la fin d'un processus fils (n'importe lequel)
`pidTermine = waitpid(pidAAttendre, NULL, 0);`	Permet d'attendre la fin d'un processus spécifique

I.7. Sémaphores (`#include ‹semaphore.h›`)

semaphore permet de créer un sac avec des jetons et de faire des synchronisations faciles (nécessite de créer plusieurs processus ou plusieurs threads).

Instruction	Utilité
`sem_t monSemaphore;` `sem_init(&monSemaphore, 0, nbJetons);`	Créer un sémaphore en initialisaant des jetons (pourra être ajouté plus tard)
`sem_wait(&monSemaphore);`	Enlève 1 jeton au sémaphore ou attend qu'un jeton soit ajouté s'il n'y a plus de jeton à retirer (le processus s'endort))
`sem_post(&monSemaphore);`	Ajoute 1 jeton au sémaphore (réveille éventuellement 1 processus au hasard qui attend un jeton)