



Du langage Java au langage C

Objectifs

Ce document s'adresse à un public d'analyste-programmeurs connaissant déjà le langage Java.

Il ne présentera donc pas les structures algorithmiques du langage C, qui sont les mêmes qu'en Java, mais il insistera sur les spécificités du C :

- notion d'interface de fonctions, et fichiers d'entête. Directive **#include**
- utilisation des bibliothèques d'entrée-sortie
- utilisation des pointeurs dans le passage de paramètres, et l'allocation dynamique de variables

Ce document n'est pas une présentation exhaustive du langage C : il donne quelques repères en s'appuyant sur le langage Java, et propose un parcours guidé du livre de Claude Delannoy : « **Le livre du C premier langage** ».

Ce que vous devez retenir de Java pour programmer en C

- les commentaires type Java, C++ sont reconnus par le C : *// Je suis un commentaire*
- Toutes les structures algorithmiques (if, for, while...), la déclaration de variables, la déclaration de fonctions et leurs appels. On signalera les différences concernant la déclaration de variable.
- Le langage C distingue les minuscules et les majuscules, comme Java et C++ (il est "case-sensitive").

Ce que vous ne pourrez pas faire en C

- *Manipuler des classes* : C n'est pas un langage orienté objet. Il ne reconnaît pas le mot clé **class**. Vous pourrez seulement manipuler des structures de données, sans fonctions membres, en les déclarant par le mot clé **struct**.
- *Surcharger les fonctions* : comme tous les compilateurs « classiques » (avant Ada, C++ et Java), le langage C ne sait pas distinguer les fonctions par leur « signature ». Des fonctions distinctes devront avoir des noms différents.
- *Charger dynamiquement des fonctions* : un programme C possède un point d'entrée unique qui est la fonction **main**. Toutes les fonctions appelées dans le déroulement du programme doivent être « linkées » avec le **main**, pour constituer un fichier exécutable, avec l'extension **.exe**.
- *Faire tourner le même exécutable sur plusieurs plate-formes matérielles* : le langage C, comme tous les compilateurs classiques, ne passe pas par un « byte-code ». Le code généré est donc destiné à un seul type de processeur.
- *Déclarer des variables n'importe où dans le code* : les variables seront déclarées en début de fonction ou en début de bloc, un bloc étant repéré par les accolades { }
- Déclarer une variable dans l'entête d'une boucle *for*

- *Utiliser les commentaires du C* : ces commentaires peuvent contenir des caractères sur une ou plusieurs lignes, qui sont délimités par **/*** et ***/**. Cette forme de commentaire est très pratique pour les entêtes de fichier et de fonction.
- *Déclarer des constantes* : le mot clé **const** du langage C correspond au **literal** de Java. Il définit une « constante typée » qui améliore la lisibilité du programme :

```
const int MAX = 100 ;
```

- Le langage C manipule également des « constantes non typées » par la directive **#define** :

```
#define MAX 100
```

On utilisera de préférence les constantes typées. A noter que **#define** effectue une simple substitution de MAX par 100, avant la compilation (toutes les lignes commençant par un # sont exécutées par le « pré-processeur » qui prépare le véritable code du programme C).

- *Allouer ou libérer de la mémoire dynamique* grâce aux opérateurs **malloc** et **free**

```
/* *****  
Fichier source      : base.C  
***** */  
  
#include <stdio.h>  
  
void main ()  
{  
    int total;          /* Définition d'une variable de type entier.  */  
  
    total = 2 + 2;      /* Appel de la fonction printf externe qui  */  
    printf("Voici la reponse : "); /* imprime ici une chaîne de caractères  */  
    printf("%d", total); /* et le résultat.  */  
}
```

- la fonction **main** doit être unique dans le programme C : c'est le point d'entrée (équivalent du **static void main** dans une classe Java)
- la fonction **printf** (équivalent de **System.out.print** en Java) permet d'afficher à l'écran tous les types scalaires (caractères, réels, entiers) et le type **string** du C (tableau de caractères terminé par 0).
- **printf** est une fonction **externe**, située dans une bibliothèque du C : bien que les prototypes ne soient pas obligatoires en C, on les indiquera toujours. Ils permettent au compilateur d'effectuer les vérifications nécessaires sur le nombre, et les types de paramètres. Ils sont obligatoires en C++.
- Les prototypes de la fonction **printf** et de nombreuses autres fonctions d'entrées-sorties sont fournis par le fichier d'entête **stdio.h** (en anglais fichier « **header** » d'où l'extension **.h**).

- Les fichiers d'entête sont de simples fichiers texte, qui sont inclus dans le fichier avant la compilation par la directive **#include**. Comme toutes les lignes commençant par **#**, **#include** est interprété par le « pré-processeur » avant la compilation C proprement dite.
- Lorsqu'on nous écrirons nos propres bibliothèques de fonctions, il faudra fournir un fichier .h que nos utilisateurs incluront dans leur programme appelant.

Exemple : **#include "Factorielle.h"**

Le nom du fichier est entouré par des guillemets et représente un chemin en relatif ou en absolu. Dans l'exemple **base.c**, les < > dans **<stdio.h>** signifie que **stdio.h** est une bibliothèque système, rangée dans un répertoire connu du compilateur.



Les types simples : p. 37 à 46



Exemples d'affichage avec la fonction printf : p. 52 à 64



Rechercher le fichier d'entête stdio.h sur votre machine. Regarder les prototypes des fonctions.

- Lire l'exemple qui suit, en vérifiant que vous reconnaissez tous les formats d'affichage utilisés :

```
/******
```

```
-- Fichier source      : formats.c
```

Quelques repères sur l'utilisation de printf

- Les caractères de contrôle en impression :

<i>line feed ou newline</i>	<i>LF</i>	<i>\n</i>
<i>horizontal tabulation</i>	<i>HT</i>	<i>\t</i>
<i>backspace</i>	<i>BS</i>	<i>\b</i>
<i>carriage return</i>	<i>CR</i>	<i>\c</i>
<i>form feed</i>	<i>FF</i>	<i>\f</i>
<i>backslash</i>	<i>\</i>	<i>\\</i>
<i>single quote</i>	<i>'</i>	<i>\'</i>
<i>bit pattern</i>	<i>ooo</i>	

\ooo ooo = une valeur octale.

Ex : \x7 --> 'BEEP'

- Les formats d'impression :

<i>char</i>	<i>-></i>	<i>%c</i>	
<i>int et char</i>	<i>-></i>	<i>%d</i>	<i>(décimal)</i>
		<i>%o</i>	<i>(octal)</i>
		<i>%x</i>	<i>(hexadécimal)</i>
		<i>%u</i>	<i>(décimal non signe)</i>
<i>float</i>	<i>-></i>	<i>%f</i>	<i>(virgule flottante + marque décimale)</i>
		<i>%e</i>	<i>(notation 'ingenieur')</i>
		<i>%g</i>	<i>(le plus court des deux précédents)</i>

**/*


```

#include <stdio.h>

void main ()
{
    int        entier1, entier2;
    char        caractere;
    float        flottant;

    caractere = 'a';          /* attention à la différence entre :          */
                               /* - une constante caractère délimitée par : ' ' */
                               /* - une chaîne de caractères délimitée par : " " */

    entier1 = 6547382;
    entier2 = 85;
    flottant = 3.14159f; /* noter le f pour indiquer le type de la constante
                               sinon le C réserve un double par défaut */

    printf (" Quelques formats d'affichage en C\n\n");

    printf (" Reel %f\n", 123.4);

    printf (" Caractere = %c \n Entier = %d \n Reel = %f \n",
            caractere, entier1, flottant);

    printf (" Reel avec format scientifique (e) = %e \n Reel avec format g = %g \n",
            flottant, flottant);

    printf (" Entier affiche en decimal = %d, octal = %o, hexa = %x\n",
            entier2,entier2,entier2);

    printf (" Reel avec virgule = %8.5f (8 chiffres dont 5 apres la virgule) \n",
            flottant);
}

```



Exercice 1

Afficher une table ASCII sur deux colonnes, en se limitant aux caractères imprimables (à partir du caractère espace, de code ascii 32)

*Lecture caractère par caractère par la fonction **getchar***

- Equivalent de la fonction **Lire.c()** de notre bibliothèque « maison » en Java
- Attention : la fonction **getchar** lit tous les caractères saisis au clavier, y compris les caractères de contrôle (par exemple <return> noté '\n')

```
/******  
  
Fichier source          : LectureBase.c  
  
*****/  
  
# include <stdio.h>  
  
// Prototypes des fonctions utilisées (fournies dans ce fichier)  
void Question();  
  
main ()  
{  
  
    int reponse;  
  
    Question ();  
  
    reponse = getchar();  
  
    if (reponse == 'N')  
  
        printf("\nAlors faites du Visual Basic !");
```

```
    else if (reponse == 'O')
        printf("\nTant Mieux");
    else
        printf("\nReponse inconnue");
}

void Question ()
{
    printf ("Aimez-vous le langage C ?(O/N)");
}
```



Exercice 2

Modifier l'exemple précédent pour qu'il boucle jusqu'à ce que l'utilisateur réponde 'O' à la question. Que se passe-t-il ?

Ecrire une fonction **Purge** qui lise tous les caractères tapés au clavier jusqu'au caractère <RETURN>. Appeler la après la saisie de votre réponse, pour corriger le problème rencontré.

Pour comprendre l'interface de la fonction de lecture **scanf** (p. 64 à 73), il faut savoir comment manipuler les adresses et les pointeurs en langage C, et comment les utiliser pour passer des paramètres en sortie.

- On appelle pointeur en C/C++ une variable qui peut exclusivement recevoir l'adresse d'une autre variable.
- Les pointeurs sont typés :

int * pentier ; // *pentier est un pointeur (*) d'int*

char * pcar ; // *pcar est un pointeur de char*

float * preel ; // *preel est un pointeur de float*

- Un pointeur doit être initialisé avec l'adresse d'une variable du type correspondant, sinon il pointe n'importe où, et son utilisation entraîne des "violations mémoire". Cette initialisation se fait avec l'opérateur d'adresse noté **&**

int i = 10; (1)

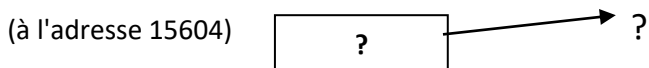
int * pentier; (2)

pentier = &i; (3) // *pentier pointe sur l'entier i*

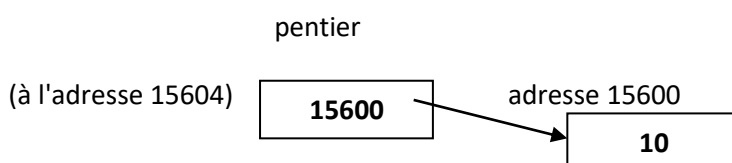
Etape (1) : déclaration de l'entier **i**



Etape (2) : déclaration du pointeur **pentier** (au départ, le pointeur contient n'importe quoi)



Etape (3) : le pointeur **pentier** pointe sur l'entier **i**



- On récupère le contenu de la variable pointée par le pointeur **p**, par la syntaxe ***p**. Dans l'exemple précédent :

printf ("%d", *pentier); affichera 10 comme **printf ("%d", i);**

***pentier = 100;** range la valeur 100 dans la variable **i**, pointée par **pentier**

Utiliser l'opérateur ***** revient graphiquement à suivre le chaînage (~~→~~) et à travailler sur le contenu de la variable pointée.

Pour les premiers exercices sur les pointeurs, on fera soigneusement les dessins des chaînages correspondant au code en C.



Chapitre IX : Les pointeurs (p. 183-188)

```

/*****

Nom du programme : pointer1.c

*****/

#include <stdio.h>

void main ()
{
    int *pentier;          // Déclaration d'un pointeur d'entier

    int x=10, y=0;         // Déclaration et Initialisation de x et y

    printf ("Au depart : x=%d y=%d\n", x, y);

    pentier = &x;          // pentier reçoit l'adresse de x

    y = *pentier;          // y := contenu de l'adresse mémoire pointée par pentier

    printf ("A la fin : x=%d y=%d\n", x, y);
}

```



Exercice 3

Faire tourner cet exemple sous Debugger : vérifier que le pointeur contient n'importe quoi à sa déclaration, repérer l'adresse de la variable x et vérifier le contenu du pointeur après l'affectation...

```

/*****

Nom du programme : pointer2.c      (version avec trace)

*****/

#include <stdio.h>

void main ()
{
    int *pentier;          // Définition d'un pointeur

    int x = 10;            // Définition d'une variable initialisée
}

```

```

printf ("Pointeur pentier:  adresse : %p, contenu %p \n\n",
        &pentier, pentier);

printf ("Variable x : adresse %p, contenu %d\n\n", &x, x);

pentier= &x;                // Initialisation du pointeur

printf ( "Apres initialisation, pointeur pentier: adresse %p, contenu %p\n",
        &pentier, pentier);

printf ( "Variable pointee par pentier: adresse %p, contenu %d\n",
        pentier, *pentier);
}

```

L'utilisation des pointeurs dans le passage des paramètres en sortie

Le langage C passe tous les types de paramètre par valeur, à l'exception des tableaux.

Soit la fonction :

```

void Afficher (int x)
{
    printf ("Je recois l'entier %d", x) ;
}

```

et son appel dans le main (extrait de code) :

```

{
    int valeur = 100;
    Afficher (valeur);
}

```

....etc..

Le C range le contenu de la variable passée en paramètre dans une **PILE** :

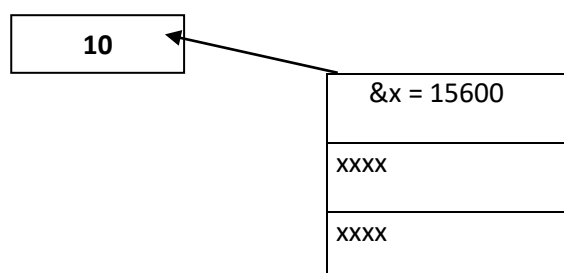
100
xx
yy

Lorsqu'elle travaille sur le paramètre x, la fonction **Afficher** travaille en fait sur la case de la PILE qui a reçu la "valeur" de la variable : ce mécanisme est parfait pour un passage de paramètre en entrée, mais ne convient pas pour un passage de paramètres en sortie, car on travaille toujours sur une **copie** de la variable passée en paramètre !

La seule solution en C pour traduire la notion de paramètres en sortie consiste à "empiler" l'adresse de la variable (au lieu de son contenu), et à récupérer cette adresse dans un pointeur, dans la fonction. Le pointeur est lui-aussi manipulé par "valeur", mais connaître la "valeur" du pointeur, c'est bien-sûr connaître l'emplacement de la variable pointée et pouvoir y accéder :

(à l'adresse 15600) variable x

PILE



Déclaration d'une fonction avec un pointeur d'entier, pour traduire un passage de paramètre en entrée-sortie sur un entier :

```
void Incrementer (int * px)
{
    * px = * px + 1;
}
```



```
}

```

Dans le corps de la fonction, on accède à la variable pointée par l'opérateur *

Appel de la fonction **Incrementer** dans le main (extrait de code) : on passe en paramètre l'adresse de la variable grâce à l'opérateur **&**

```
int valeur = 100;

Incrementer (&valeur);

// valeur vaut maintenant 101
```

```

/*****

Fichier source      : parametres.c

Passage de parametres par valeur et par adresse

*****/

#include <stdio.h>

// définitions des prototypes des fonctions

void f_parvaleur (int i);

void f_paradresse (int *pointeur_i);

void main()
{
    int argum = 1;

    printf("\nProgramme principal : argument %d \n", argum);
    printf("\nExecution de f_parvaleur(argum) \n");
    f_parvaleur( argum);

    /* argum ne va pas être modifié dans la programme principal */

    printf("Programme principal : argument=%d\n", argum);
    printf("\nExecution de f_paradresse( &argum) \n");
    f_paradresse( &argum);

    /* argum est modifié */

    printf("Programme principal : argument=%d\n", argum);
}

void f_parvaleur (int i)
{
    i = i + 10;

    printf(" Valeur dans f_parvaleur : parametre=%d\n", i);

    /* i est modifié uniquement dans la fonction */
}

```

```

}

void f_paradresse (int *pointeur_i)
/* declaration d'un pointeur sur i */
{
    *pointeur_i = 100;
    printf(" Valeur dans f_paradresse : parametre=%d\n", *pointeur_i);
    /* i est modifié dans le programme principal */
}

```



Chapitre IX : Les pointeurs (p. 189-190)



Exercice 4

Ecrire une fonction **Puissance** avec l'interface suivante, et son programme de test

fonction **Puissance**

```

( Entrée  operande : reel      // nombre à élever à une puissance entière
  Entrée  exposant : entier    // exposant
  Sortie  resultat  : reel      // resultat du calcul s'il est possible
): boolean // vrai si calcul possible

```

La fonction **Puissance** élève le réel **operande** à la puissance **exposant** (positif ou nul) et range le résultat dans **resultat**.

Cas d'erreur : exposant négatif, overflow.

Ce détour par les pointeurs va nous permettre de bien comprendre l'interface de la fonction de lecture clavier : **scanf**

Cette fonction reçoit deux paramètres : un format de lecture (avec les mêmes valeurs que **printf** : **%c**, **%d**, **%f** ...) et **l'adresse de la variable** où **scanf** devra ranger son résultat (paramètre en sortie)

Attention à la différence entre les appels de **printf** et **scanf** : **printf** attend en paramètre la **valeur** de la variable à afficher (paramètre en entrée) ; **scanf** attend en paramètre **l'adresse** de la variable à modifier.

```
/* **** */
Fichier source      : Testscanf.c
Mise en oeuvre de scanf
**** */

#include <stdio.h>

main()
{
    int entier;
    char caractere;
    char chaine[30]; /* Tableau de 30 caractères */

    printf ("Mise en oeuvre du scanf\n\n");

    printf ("Entrez un entier :");
    scanf ("%d", &entier);

    /* Instruction servant a vider le buffer, sinon la saisie du */
    /* caractere prendra le retour chariot suivant l'entier */
    /* (c'est notre fonction Purge fournie par le langage C !) */
    fflush ( stdin );
}
```

```

printf ("Entrez un caractere :");

scanf ("%c", &caractere);

printf ("Entrez une chaine :");

scanf ("%s", chaine);

printf ("\nReaffichage des valeurs saisies\n\n");

printf("\n\t\tEntier entre      : %d\n", entier);

printf("\t\tCaractere entre : %c\n", caractere);

printf("\t\tChaine entree   : %s\n", chaine);

}

```

Remarque importante : dans le dernier appel de **scanf**, le paramètre **chaine** n'est pas précédé par l'opérateur d'adresse **&**. Ce n'est pas une exception à la règle car **chaine** est un tableau : le nom de tableau en langage C, sans les **[]** désigne toujours l'adresse de la première case du tableau .

2 notations équivalentes pour désigner l'adresse de début d'un tableau **tab** : **tab** ou **&tab[0]**



Exercice 5 : reprise de la Calculatrice en langage C

Nous en savons maintenant assez pour réécrire entièrement la calculatrice analysée dans le document **Découverte de la programmation en Java** : p. 47 à 52

On profitera de cet exercice pour découvrir la "compilation séparée" en langage C : les fonctions **SaisirOperande**, **SaisirOperateur**, et **Afficher** seront regroupées dans un fichier **Dialogue.c**, fourni avec son fichier d'entête **Dialogue.h**. La fonction **EffectuerCalcul** sera placée dans un fichier **Calculs.c** fourni avec son fichier d'entête **calcul.h**. Le programme principal **Calcullette.c** utilisera les deux bibliothèques de fonctions, en incluant les fichiers d'entête **Dialogue.h** et **Calculs.h**.

On s'inspirera de l'exemple suivant (reprise du programme **parametre.c** en compilation séparée) :

```

/*****

Fichier source          : fonctions.c

```

```

    Bibliothèque de fonctions appelées par le programme Principal.c

    Utilisation : démonstration de compilation séparée

    **** */

#include <stdio.h>

#include "fonctions.h"

void f_parvaleur (int i)
{
    i = i + 10;

    printf(" Valeur dans f_parvaleur : parametre=%d\n", i);
}

void f_paradresse (int *pointeur_i)
{
    *pointeur_i = 100;

    printf(" Valeur dans f_paradresse : parametre=%d\n",
           *pointeur_i);
}

```

Dans le fichier .C de la bibliothèque, on inclut le fichier d'entête, pour vérifier la cohérence entre les prototypes déclarés dans l'entête et les interfaces des fonctions correspondantes dans le fichier .C

```

/*****

Fichier d'entête           : fonctions.h

    Prototypes des fonctions du fichier fonctions.c

    **** */

void f_parvaleur  (int i);

void f_paradresse (int *pointeur_i);

```

```

/*****

Fichier source           : Principal.c

Utilisation :

- démonstration de "compilation séparée"

- Ce fichier doit être linké avec la bibliothèque de fonctions

```

fonction.c

- le fichier d'entête doit être présent dans le répertoire du projet.

```
***** */
#include <stdio.h>
// Fichier d'include avec les prototypes des deux fonctions
#include "fonctions.h"
// inclut les prototypes des fonctions appelées
void main()
{
    int argum = 1;

    printf("\nProgramme principal : argument %d \n", argum);
    printf("\nExecution de f_parvaleur(argum) \n");
    f_parvaleur( argum);
    /* argum ne va pas être modifié dans la programme principal */

    printf("Programme principal : argument=%d\n", argum);
    printf("\nExecution de f_paradresse( &argum) \n");
    f_paradresse( &argum);
    /* argum est modifié */
    printf("Programme principal : argument=%d\n", argum);
}
```

Les tableaux en langage C

Contrairement au langage Java, le langage C gère les tableaux comme des **variables statiques**, de longueur fixe définie à la déclaration.

Rappel : la classe de compilation "statique" comprend toutes les variables dont la taille et l'emplacement sont définis à la compilation. Ces variables existent du début à la fin du programme.

`char tab [1000];` // déclare un tableau à une dimension de 1000 caractères, indicés de 0 à 999

~~`int MAX = 1000;`~~

`char tab[MAX];` // ne compile pas, car le contenu de la variable MAX n'est pas connu au moment de
// la compilation : le compilateur tente alors d'allouer une zone de 0 octet

~~`const int MAX = 1000;`~~

`char tab[MAX];` // paraît logique, mais ne compile pas non plus, car les constantes typées ne sont pas
// évaluées à la compilation

`#define MAX 1000`

`char tab[MAX];` // Compile : car le pré-processeur remplace MAX par 1000, avant la compilation



Chapitre VII : Les tableaux (p. 128-154)

Les chaînes de caractère (*string*)

Une chaîne, au sens du langage C, est un simple tableau de caractères, dont la partie utile est terminée par l'octet 0 (Attention : pas le caractère '0' de code ASCII 48).

A	F	P	A	0
---	---	---	---	---

La bibliothèque **string.h** fournit des fonctions de manipulation de chaîne : **strcmp** pour comparer deux chaînes, **strcpy** pour recopier une chaîne sur une autre, **strcat** pour "concaténer" deux chaînes (recopier une chaîne à la suite d'une autre), **strlen** pour connaître la longueur d'une chaîne ...

On peut transformer facilement un tableau de caractères existant en chaîne de caractères, en recopiant le terminateur 0 à la suite des caractères utiles. Toutes les fonctions de la bibliothèque **string.h** manipulent correctement ce terminateur.

Attention Le "bug" le plus répandu sur les **string** en C, qui provoque des "violations mémoire", consiste à perdre le terminateur 0 : dans ce cas, les fonctions de manipulation de string parcourent la mémoire jusqu'à trouver un terminateur.



Chapitre X : Les chaînes de caractères (p. 194-211)



Exercice 6 : Manipuler des tableaux de string

Le logiciel demandé :

- stockera une liste de noms saisis au clavier (*=> dans un tableau de string*)
- triera la liste par ordre alphabétique (*=> tri par remontée des bulles*)
- réaffichera la liste une fois triée.

Pour réaliser cet exercice, on adaptera l'exemple **tlettres.java** (**Découverte de la programmation en Java**, chap. 6), en manipulant des tableaux de **string** à la place des tableaux de caractères.

Pour structurer nos données et rendre le programme plus lisible, nous utiliserons la directive **typedef** du langage C, qui définit un nouveau type utilisateur, à partir de types utilisateur existants ou de types prédéfinis fournis par le langage C :

```
// LGNOM : nombre maximum de caractères d'un nom

#define LGNOM 30

// Nom est un nouveau type : n'importe quel tableau de LGNOM cases de type char
typedef char Nom [LGNOM];

// LGLISTE : nombre maximum de noms dans la liste

#define LGLISTE 20

// Liste est un nouveau type : n'importe quel tableau de LGLISTE cases de type Nom
```

```
typedef nom Liste [LGLISTE];
```

```
// Deux listes de noms : les amis et les ennemis
```

```
Liste amis, ennemis;
```

Avec cette méthode, on peut décrire des données complexes, sans jamais aboutir à des syntaxes "horribles", comme on les rencontre parfois dans les exemples de C "historiques".

On respectera des conventions simples d'écriture :

- les constantes tout en majuscules,
- les identificateurs de type commencent par une lettre majuscule,
- les identificateurs de variables commencent par une lettre minuscule.



Conseils pour la réalisation

- *Utiliser la compilation séparée pour travailler proprement :*
 - ⇒ les fonctions d'entrée-sortie **Lecture** et **Affichage** seront regroupées dans un fichier **Dialogue.c**,
 - ⇒ les fonctions **TriAlphabetique** et **Permuter** dans un fichier **Tri.c**,
 - ⇒ la définition des types de données (ci-dessus) dans un fichier **Donnees.h** qui ne contiendra que des définitions de constantes et des directives **typedef**
- *Commencer par définir :*
 - ⇒ les structures de données (donc le fichier d'entête **Donnees.h**),
 - ⇒ toutes les interfaces des fonctions utilisées (donc les fichiers d'entête **Dialogue.h** et **Tri.h**).
 - ⇒ faites valider vos interfaces, avant de commencer le codage des corps de fonction.
- *Appliquer une vraie méthode de construction de programme :*

- ⇒ tester entièrement chaque fonction séparément avant de commencer la suivante
- ⇒ dans un ordre logique de réalisation : **Lecture, Affichage, Permuter, TriAlphabétique**

- *Ne pas perdre de temps avec la présentation : ce n'est pas l'objectif !*

Equivalence tableau <--> pointeur

Un tableau **int tab[6]**; ne peut se manipuler par son nom **tab**, pour des affectations ou des comparaisons entre tableaux.

En effet, **tab** représente l'adresse du tableau (l'adresse où est rangé le 1^{er} des 6 entiers) :

tab ⇔ **&tab[0]**

Ainsi le nom d'un tableau pointe sur le 1^{er} élément de ce tableau, et :

tab[0] ⇔ ***tab**

tab[i] ⇔ ***(tab + i)**

Attention à la précedence des opérateurs et aux parenthèses :

- l'opérateur ***** est plus prioritaire que l'opérateur **+**.
- ***(tab + i)** avance d'abord de **i** cases à partir de l'adresse **tab**, puis prend le contenu de la case d'arrivée.
- ***tab + i** ajoute la valeur **i** au contenu de la case située à l'adresse **tab**.

Puisqu'il représente l'adresse du tableau, le nom du tableau **tab** peut servir à initialiser un pointeur de même type que les éléments du tableau :

```
int * pointeur = tab;
```

alors

***(pointeur+i)** vaut **tab[i]** et **pointeur++** avance d'une case dans le tableau dynamique.

Attention : l'addition sur un pointeur et l'incrémentation se font du nombre d'octets adéquat, selon le type de données pointées :

```
char * p; // p++ avance d'un octet
```

mais

```
int * p; // p++ avance de 4 octets...
```

Ces équivalences que l'on trouve dans tous les livres de C ne vont pas nous servir à gérer des tableaux comme des pointeurs, pour obscurcir la programmation : c'est idiot ! Mais au contraire à retrouver l'équivalent des tableaux dynamiques en Java : en allouant dynamiquement de la mémoire, on pourra la parcourir plus simplement en utilisant le pointeur avec la syntaxe des tableaux : à la place de ***(pointeur+i)** nous écrirons toujours **pointeur[i]**

Pointeurs et allocation dynamique de mémoire

Utiliser la bibliothèque **malloc.h** pour gérer les espaces mémoire dynamiques. Cette bibliothèque fournit les fonctions **malloc** et **free**.

```
void *malloc ( size_t size );
```

où *size* est le nombre d'octets à allouer.

```
void free ( void *memblock );
```

où *memblock* est un pointeur vers la zone mémoire à libérer

```

/* Fichier TestMalloc.C

    Ce programme alloue de la mémoire avec malloc, puis libère la mémoire avec free.

    La mémoire est utilisée comme tableau dynamique de double (réels double précision)
    Le nombre de double à allouer est saisi au clavier.

*/

#include <stdio.h>
#include <malloc.h>

void main( void )
{
    double * ptab; // pointeur d'entier (<=> tableau dynamique d'entiers)
    int nb;         // nombre d'entiers demandés

    // 1° lecture du nombre d'entiers à allouer
    printf ("Nombre d'entiers a allouer :");
    scanf ("%d", &nb);

    // 2° allocation de la zone mémoire
    // Attention au calcul : malloc n'attend pas le nombre de cases du tableau
    // mais un nombre d'octets

    ptab = malloc ( nb * sizeof (double) ); (1)

    if (ptab == NULL)
        printf( "Memoire disponible insuffisante\n" );
    else
    {
        int i ; // indice de remplissage et de relecture du tableau

        printf( "Memoire allouee correctement\n" );

        // 3° On remplit (pour l'exemple) chaque case avec le carré de son indice

```

```

    for (i = 0 ; i < nb ; i++)
        ptab[i] = i * i ;


    // 4° Réaffichage du tableau
    for (i = 0 ; i < nb ; i++)
        printf ("Case %d : %f\n", i, ptab[i] );

    // 5° Libération de la mémoire allouée

    free (ptab); (2)

    printf( "Memoire liberee\n" );
}
}

```

(1)  Dans l'appel de **malloc**, toujours passer le nombre d'octets à allouer, quelque soit le type de la donnée à allouer. Dans le cas d'un tableau de caractères, ce nombre est le même que le nombre de cases du tableau. Dans tous les autres cas, il faut faire faire le calcul au compilateur : **nb_case * taille_case**

Pour évaluer la taille de la case, utiliser la macro **sizeof** qui retourne la taille en octets d'une variable ou d'un type.

(2) Libérer correctement la mémoire par **free**, dès qu'elle ne sert plus dans le programme (pas seulement à la fin du programme !). De nombreux bugs viennent d'une accumulation de mémoire dynamique non libérée qui finit par saturer le PC.

||

Tableau de Pointeurs et Pointeur de pointeurs

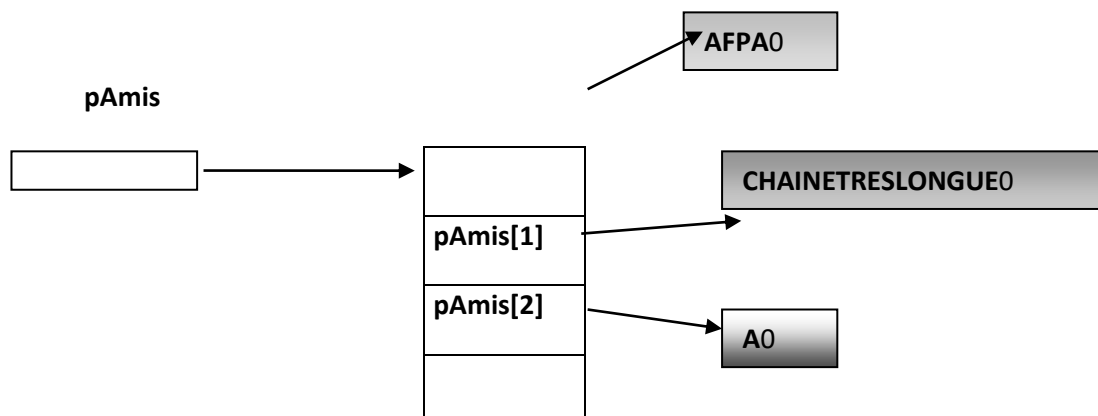
Dans une application qui manipulerait un grand nombre de données, l'exercice 6 serait considéré comme beaucoup trop gourmand en mémoire : il gaspille la mémoire par la taille du tableau (que l'on ne remplit jamais complètement), et par la taille des noms (tableau de caractères de taille fixe, indépendante de la longueur effective du nom). L'exercice suivant va donc consister à optimiser le programme sur ces deux aspects :

- le nombre de noms à gérer sera choisi par l'utilisateur au début du programme ;
- chaque nom entré sera stocké avec sa vraie taille, sans gaspillage de mémoire (en fait, sa taille + 1 pour le terminateur de string).

En règle générale, lorsque l'on fait du système (applications industrielles, logiciels de base...), on doit être économe en mémoire et renoncer à l'allocation statique : tout se fait dynamiquement, à l'aide de pointeurs, et on prend soin de libérer la mémoire dès qu'elle n'est plus utilisée.

Comme l'exercice qui suit est plus technique, nous allons partir d'un exemple proche, écrit dans un C "brut de fonderie", qui nous fournira les éléments indispensables. Il faudra ensuite l'oublier, et réécrire l'ensemble proprement, en découpant en fichiers et fonctions...

Schéma des structures de données utilisées (suivre l'exemple pages 19-20)



- (1) Le pointeur **pAmis** de type **PListe** permet de gérer un tableau dynamique de pointeurs de **char**.
- (2) Chaque case **pAmis[i]** est elle-même un pointeur de caractères (type **char ***) qui permet d'allouer une chaîne de caractères de longueur variable.

Problèmes d'analyse rencontrés :

- A la lecture de la chaîne, on ne sait pas encore combien l'utilisateur va entrer de caractères. Quel nombre de caractères faut-il allouer par **malloc** ?

⇒ La solution la plus simple consiste à utiliser une chaîne tampon pour les lectures, de la taille du nom le plus grand : **(3)** le programme commencera par demander cette taille à l'utilisateur, puis allouera le tampon à taille + 1 (toujours penser au 0 terminateur de string) **(4)**

- Comment éviter que l'utilisateur entre trop de caractères à la lecture de la chaîne ?

Si l'on utilise simplement un format de lecture "%s", l'utilisateur peut entrer autant de caractères qu'il le veut. Les caractères sont rangés à la suite les uns des autres, et peuvent déborder du tableau alloué. Solution pour un nombre de caractères constant : "%20s" limite la saisie à 20 caractères.

⇒ **(5)** Un format est une chaîne comme une autre : nous commençons donc par construire le format de lecture (variable **fmtampon**) en fonction du nombre de caractères maximum choisi par l'utilisateur.

⇒ On notera l'utilisation de la fonction **sprintf** qui permet de formater des string en mémoire (de variable à variable) sans sortie écran :

```
sprintf ( frtampon, "%%%ds", lgNom);
```

Les paramètres 2 et 3 correspondent aux deux paramètres de la fonction **printf** (format d'affichage "%%%ds", et variable à formater **lgNom**). Le premier paramètre (**frtampon**) reçoit le résultat du formatage.

Dans le format "%%%ds", le redoublement du % veut dire qu'il ne s'agit pas du caractère de contrôle mais du vrai caractère %. Le troisième % est interprété comme caractère de contrôle avec **d** : **%d**. Le **s** final est considéré comme un caractère normal. Sortie formatée : **% lgNom s**

Exemple : **%30s** pour *lgNom* = **30**

(6) Le format **frtampon** peut ensuite être utilisé comme un format statique, dans une lecture clavier par **scanf** (**frtampon, pTampon**);

Grâce au nombre de caractères indiqué dans le format, la lecture est sécurisée : les caractères en excès seront automatiquement tronqués par la fonction **scanf**.

(7) Une fois la lecture effectuée dans le tampon, le programme alloue une nouvelle chaîne dans la liste, de taille : nombre de caractères lus au clavier + 1

```
pAmis[iListe] = (PNom) malloc (strlen (pTampon)+1);
```

A noter les "**casting**" (conversions de type) systématiques sur les pointeurs : la fonction malloc nous renvoie un pointeur non typé (**void ***) que l'on transforme en pointeur de char : (**char ***) noté (**PNom**). En langage C, le compilateur n'exige pas les castings, mais il faut les faire par sécurité. En C++, ils sont requis par le compilateur.

(8) Pour libérer les structures de données, on suit l'ordre inverse de l'allocation : on libère d'abord chaque chaîne pointée par une case du tableau, avant de supprimer le tableau de pointeurs lui-même **(9)**

```
for ( iListe = 0; iListe < lgListe ; iListe ++)
```

```
free (pAmis[iListe]);
```

free (pAmis);

(10) Purger les caractères restant dans la ligne après chaque appel à **scanf**, pour ne pas fausser la logique du programme : **fflush (stdin);**

```

/*
    Fichier : tabPointeurs.c

    Ce programme crée un tableau dynamique de pointeurs de string,
    à la taille demandée par l'utilisateur, et il stocke chaque string
    avec le nombre exact d'octets saisis
*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>    // pour les fonctions malloc et free
#include <string.h>

// Type utilisateur : pointeur de caractères
// ( <==> tableau dynamique de caractères)
typedef char * PNom;

// Type utilisateur : pointeur de liste de noms
// <=> tableau dynamique de noms
typedef PNom * PListe;

void main()
{
    int lgListe;           // nombre de noms stockés
    int iListe ;           // indice du tableau dynamique de noms

    PListe pAmis; (1) // pointeur de la liste de noms (ou nom du tableau
                        // dynamique )

    int lgNom;             // longueur maximale d'un nom (utilisé pour créer
                        // la chaîne tampon)

```

```

char * pTampon;          // chaine dynamique "tampon" utilisé pour lire
                          // au clavier, avant rangement dans la liste de noms

char frtampon[6];        // format de lecture du tampon

printf ("*** Gestion d'une liste de noms (22/01/2002 v1.0) *** \n\n");

// Lecture du nombre de noms à gérer

printf ("Nombre de noms a gerer ?");

scanf ("%d", &lgListe);

fflush (stdin); // purge du buffer clavier

// allocation du tableau dynamique

// ATTENTION : malloc attend la taille de la mémoire en octets

// Pour obtenir cette taille, on multiplie le nombre de noms
// demandés, par la taille d'un pointeur PNom (directive sizeof)

// NOTER aussi le CASTING de pointeur (void *) en pointeur typé
//   PListe

pAmis = (PListe) malloc (lgListe * sizeof (PNom) );

// Lecture de la taille maximale d'un nom

// et allocation du tampon

printf ("Taille maximale d'un nom dans la liste ?");

scanf ("%d", &lgNom); (3)

fflush (stdin);

// Taille du tampon : ajouter 1 à la longueur maximale du nom

// pour ranger le caractère terminateur 0

pTampon = (PNom) malloc (lgNom+1); (4)

```

```
// Préparation du format de saisie du tampon : un format est une chaîne  
// comme une autre que l'on peut préparer dynamiquement
```

```
sprintf ( frtampon, "%%ds", lgNom);  
printf ("Format de lecture :%s", frtampon); (5)
```

```
// 4) allocation et saisie des noms
```

```
for ( iListe = 0; iListe < lgListe ; iListe ++)
```

```
{
```

```
printf ("Nom a stocker ?");
```

```
// lecture de la chaîne dans le tampon
```

```
scanf (frtampon, pTampon); (6)
```

```
fflush (stdin); (10)
```

```
// on crée dynamiquement dans la liste une chaîne de
```

```
// même taille que le nombre de caractères lus + 1
```

```
pAmis[iListe] = (PNom) malloc (strlen (pTampon)+1); (7)
```

```
(2)
```

```
// et on recopie le tampon sur la nouvelle chaîne
```

```
strcpy (pAmis[iListe], pTampon);
```

```
}
```

```
// réaffichage de la liste
```

```
printf ("\n ** Reaffichage de votre liste **** \n");
```

```
for ( iListe = 0; iListe < lgListe ; iListe ++)
```

```
printf ("Nom %d : %s \n", iListe, pAmis[iListe] );
```

```
// Libération de toutes les chaines dynamiques

for ( iListe = 0; iListe < lgListe ; iListe ++ )

    free (pAmis[iListe]); (8)


// libération du tableau de pointeurs

free (pAmis); (9)


// libération du tampon de saisie

free (pTampon);

}
```



Exercice 7 : optimisation de l'exercice 6

Sans casser la structure du programme (interfaces des fonctions, découpage en fichiers), on l'optimisera pour économiser le plus possible la mémoire.

Le mot clé **struct** en langage C permet de regrouper des données pour constituer une nouvelle entité avec une signification précise (ne jamais faire un fourre-tout).

Vous allez retrouver certains des réflexes acquis en manipulant les "classes" Java. Deux différences importantes : contrairement aux classes, les structures ne contiennent que des "données membre", et jamais de "fonctions membre" ; toutes les données membre sont par défaut "public" (il n'y a pas d'encapsulation en langage C).

Il y a plusieurs syntaxes possibles pour déclarer une structure. Limitons-nous à la plus propre : nous définirons toujours un nouveau type utilisateur par la directive **typedef** :

```
typedef struct
{
    Libelle lib;

    Etat     status;

    double   val;

} Descripteur ;      // nouveau type utilisateur, manipulable comme int, char...

Descripteur temp1;  // temp1 est une variable du type Descripteur
```

L'accès aux différents champs se fait comme en Java, par l'opérateur **.**

On peut recopier tous les champs d'une structure sur une autre de même type, par l'opérateur d'affectation **=** (qui ne marche pas pour les tableaux !).

Un pointeur de structure se déclare et s'utilise de la même manière qu'un pointeur sur une donnée scalaire :

```
Descripteur * pPression;

pPression = (Descripteur *) malloc ( sizeof (Descripteur) );
```


Pour accéder à un champ d'une structure par l'intermédiaire d'un pointeur, il faut d'abord accéder au contenu de la structure (par l'opérateur *), et prendre le champ demandé (par l'opérateur .), ce qui donne la syntaxe, assez lourde :

```
(*pPression).val = 23.5;
```

A noter les parenthèses qui sont nécessaires, car l'opérateur . est prioritaire sur l'opérateur * : *pPression.val se référerait au contenu de la variable pointée par le pointeur val dans la structure pPression.

Les langages C et C++ fournissent donc une expression simplifiée pour accéder aux données membre d'une structure par l'intermédiaire d'un pointeur : opérateur ->

```
pPression->val = 23.5;
```



Chapitre XI : Les structures (p. 213-224)

Résumons ces quelques points par un exemple :

```
/******  
  
Programme      mesure.c  
  
On gère un ensemble de mesure. Une mesure a pour caractéristique :  
1° un libellé  
2° l'état du capteur : s'il est en panne,  
   la mesure est sans signification  
3° la valeur de la mesure.  
*****/  
  
#include <stdio.h>  
  
#include <string.h>  
  
#include <malloc.h>
```

```

typedef char Libelle[20];

typedef enum { HS, OK } Etat; (1)

typedef struct
{
    Libelle    lib;
    Etat status;
    double    val;
} Descripteur;

void Afficher (char msg[], Descripteur des);

void main ()
{
    // déclaration d'une variable de type "Descripteur"
    Descripteur capteur1 ;

    // Pointeur vers un descripteur
    Descripteur * pCapteur2 ;

    #define MAX 3

    // Tableau de structures
    Descripteur tabDescri [MAX];

    // Tableau de pointeurs de structures
    Descripteur * pDescri [MAX];

    printf("*** Demo sur les structures ***\n");

    // initialisation des champs (Notation avec le .)
    strcpy (capteur1.lib,"PRES1" );

    capteur1.status = OK; (2)

```

```

capteur1.val = 23.90;

// Affichage complet de capteur1
Afficher ("C1", capteur1);

// allocation dynamique de pPression2
pCapteur2 = (Descripteur *) malloc (sizeof (Descripteur) ); (3)

// A noter, la flèche pour accéder à un champ d'une structure
// par son pointeur
strcpy (pCapteur2->lib,"TEMP1" );
pCapteur2->status = HS;
// Notation synonyme
(*pCapteur2).val = 100.00;

// Affichage complet du capteur
Afficher ("C2", *pCapteur2); (4)

// recopie complète d'un enregistrement sur un autre, et réaffichage
*pCapteur2 = capteur1; (5)
Afficher ("C2", *pCapteur2);

// initialisation et réaffichage d'un tableau de structures
{
    int i;
    for (i = 0; i < MAX; i ++)
    {
        sprintf (tabDescri[i].lib,"PRES%d", i ); (6)
        tabDescri[i].status = OK;
    }
}

```

```

        tabDescri[i].val = 100.0 + i;
    }

    for (i = 0; i < MAX; i ++)
        Afficher ("tabDescri", tabDescri[i]);
}

// initialisation et réaffichage d'un tableau de pointeurs de structures
{
    int i;

    for (i = 0; i < MAX; i ++)
    {
        pDescri[i] = (Descripteur *) malloc (sizeof (Descripteur) );
        sprintf (pDescri[i]->lib,"TEMP%d", i );
        pDescri[i]->status = HS;
        pDescri[i]->val = 1000.0 + i;
    }

    // réaffichage

    for (i = 0; i < MAX; i ++)
        Afficher ("pDescri", *pDescri[i]);

    // libération de la mémoire dynamique par free

    for (i = 0; i < MAX; i ++)
        free (pDescri[i]);    (7)
}

}

void Afficher (char msg[], Descripteur des)
{
    Libelle etats[2] = {"HS", "OK" }; (8)
}

```

```
printf("\nCAPTEUR : %s\t", msg);  
printf ("Libelle:%s\t", des.lib);  
printf ("Etat   :%s\t", etats[des.status] ); (9)  
printf ("Valeur :%f\n", des.val);  
}
```

- (1) Avec une syntaxe proche de celle des structures, on peut définir en langage C et C++, un type "énuméré" : c'est une suite de constantes représentées par des mnémoniques (*HS*, *OK*) qui peuvent être manipulées telles quelles dans le programme.

En interne, le compilateur considère ces constantes comme des entiers : *HS* vaut 0, *OK* vaut 1. L'utilisation de types énumérés améliore la lisibilité des programmes.

- (2) A noter l'utilisation de l'opérateur `•` pour accéder au champ *status* de la structure *capteur1* et l'utilisation de la constante *OK* définie dans le type énuméré *Etat*.

Attention : ne pas confondre la constante *OK* du type énuméré (qui vaut 1) et la string "*OK*"

- (3) Allocation dynamique d'une variable de type *Descripteur*. A noter :

- l'utilisation de *sizeof* sur le type *Descripteur* pour allouer le bon nombre d'octets,
- le **casting** qui transforme le pointeur (*void **) renvoyé par *malloc* en pointeur de *Descripteur* (*Descripteur **)

- (4) Noter l'utilisation de l'opérateur `*`, pour récupérer le contenu complet de la structure pointée par *pCapteur* : notre fonction *Afficher* attend un passage de structure par valeur, et non pas par adresse.

Il serait plus judicieux par la suite de faire des passages par adresse, pour éviter d'encombrer la Pile.

- (5) **Idem** : recopie complète d'une structure sur une autre. Accès direct à la variable *capteur1*, accès à la variable dynamique par l'opérateur `*` sur le pointeur *pCapteur2*.

- (6) **Rappel** : utilisation de *sprintf* pour formater une string en mémoire : *PRES1*, *PRES2* ...

- (7) Libération de la zone de mémoire pointée par la case numéro *i* du tableau *pDescr*

- (8)** Méthode souvent utilisée pour afficher des types énumérés : il n'est pas intéressant d'afficher *0* pour *HS* et *1* pour *OK* (incompréhensible !).

Les types énumérés seront donc utilisés comme indices d'un tableau de *string* : à l'énuméré *OK* (entier 1) correspondra la *string* "*OK*" qui pourra s'afficher en clair.

- (9)** Affichage de la string dans le tableau *etats*, à l'indice correspondant à l'énuméré *status*.



Exercice 8 : Gestion d'une liste de mesures

On commencera par optimiser l'exemple ci-dessus en remplaçant le tableau de pointeurs, par un tableau dynamique de pointeurs (c'est-à-dire un "pointeur de pointeurs").

On pourra s'inspirer de l'exercice précédent, pour la gestion du tableau dynamique : on manipule maintenant des pointeurs de structure au lieu de pointeurs de **char**...

Voir aussi l'exemple **tabMesures.java** dans le chap. 7 du document **Découverte de la Programmation en Java**.

On ajoutera plusieurs fonctionnalités : **création**, **suppression**, **modification** et **visualisation** d'une fiche capteur, référencée par son indice dans le tableau.

Conseils techniques :

- tous les pointeurs du tableau seront d'abord initialisés à **NULL** (constante = 0 qui caractérise un pointeur non initialisé).
- Lors d'une suppression, la fiche sera détruite par **free**, et le pointeur sera remis à **NULL**.
- Lors d'une demande de visualisation ou d'une modification de fiche par son indice, le programme vérifiera d'abord son existence (case du tableau non égale à **NULL**).

On proposera les différentes fonctionnalités par une interface en mode texte, la plus simple possible : ne pas se préoccuper de présentation. On repartira de cette maquette pour les exercices sur les fichiers.

Notes sur le cahier des charges

Pour le langage C, un fichier est une suite d'octets sur disque, sans structure particulière. On peut manipuler un fichier, en entrée-sortie "bufferisée", par la bibliothèque **stdio.h**. Comme en Java, il faut d'abord "ouvrir" un fichier, avant de pouvoir l'utiliser, et il faut le "fermer" en fin d'utilisation.

1) Ouverture de fichier : `fic = fopen ("mesure.dat", "rb");`

- **"mesure.dat"** est le nom sur disque du fichier à créer ou à relire
- **fic** est une variable du type **FILE *fic**
- **"rb"** est le mode d'ouverture. ("**r**" pour "read", "**w**" pour "write", "**b**" pour fichier binaire). Si l'on ne précise pas le mode "**b**", le fichier sera considéré comme un fichier texte.

2) Manipulation de fichiers :

Le C fournit des fonctions de lecture et d'écriture de base qui ne réalisent pas de conversion (**fwrite** et **fread**) et des fonctions de lecture et d'écriture formatées : **fscanf**, **fgets** et **fprintf**, **fputs**.

Règle simple : ne pas mélanger les genres, même si le langage tolère tout. Il faut considérer un fichier soit comme un fichier texte constitué de lignes (=> **fonctions de lecture et d'écriture formatées**) soit comme un fichier binaire découpé en blocs (**fwrite** et **fread**).

On peut se déplacer en accès direct dans un fichier découpé en blocs par la fonction **fseek** et connaître la position courante du pointeur de fichier par la fonction **ftell**.

En relecture séquentielle, on peut tester la fin de fichier par la fonction booléenne **feof** (fic)

Attention : en C, cette fonction ne renvoie VRAI qu'une fois la fin de fichier dépassée (après un échec sur une lecture).

3) Fermeture du fichier :

par la fonction **fclose(fic);**

1) Les fichiers binaires d'octets : *copie.c*

Cet exemple manipule des fichiers binaires, en les considérant comme de simples suites d'octets. Il recopie un fichier source sur un fichier cible, dont les noms sont passés en paramètre.

Utilisation : **copie test.exe test2.exe**

2) Les fichiers texte : *texte.c*

Cet exemple crée un fichier texte, à partir de lignes lues au clavier. Le fichier peut être imprimé ou édité sous **Notepad**.

3) Les fichiers binaires structurés : *ficdirecte.c*

Même si le langage C considère tout fichier comme une suite d'octets, on peut manipuler un fichier par blocs, grâce aux fonctions **fwrite** et **fread**, et se placer en accès direct dans le bloc de son choix, par **fseek**. On obtient alors l'équivalent d'un tableau sur disque. Cet exemple reprend les structures de données de l'exemple *mesure.c*.

```
/******  
  
Programme      copie.c  
  
Copie binaire d'un fichier source vers un fichier cible  
  
Le fichier source et le fichier cible sont passés en paramètres  
du programme : argv[1] et args[2]  
  
*****/  
  
#include <stdio.h>  
  
#include <process.h>
```

```

void main (int argc, char *argv[])
{
    FILE * source;    // handler du fichier source

    FILE * cible;     // handler du fichier cible

    // argc (argument counter) reçoit le nombre de paramètres
    // passés au programme : 3
    // le premier paramètre est le nom de l'exécutable

    if (argc != 3)
    {
        printf ("** Syntaxe : copie source cible" );
        exit (1);
    }

    // ouverture en lecture du fichier source, au format binaire
    // le nom du fichier source est le premier paramètre (argv[1] )

    source = fopen ( argv[1], "rb");

    if ( source == NULL)
    {
        printf ("Fichier source inexistant ! ");
        exit (2);
    }

    // création du fichier cible, en binaire (option b)
    // le nom du fichier source est le deuxième paramètre (argv[2]

    cible = fopen (argv[2], "wb");

    // relecture du fichier source octet par octet et recopie
    // dans le fichier cible

    {
        char octet; // octet lu dans la source, copié dans la cible

        int numlu;  // nombre d'octet lu : doit être égal à 1
    }

```

```

// lecture d'un octet dans le fichier source

numlu = fread (&octet, 1, 1, source);

while (numlu == 1)
{
    // écriture d'un octet dans le fichier cible

    fwrite ( &octet, 1, 1, cible);

    // lecture de l'octet suivant

    numlu = fread (&octet, 1, 1, source);

}

}

// fermeture des deux fichiers

fclose (source);

fclose (cible);

}

```

```

/*****

Programme      Texte.c

Saisie et réaffichage d'un fichier texte. Le fichier peut
être lu par Notepad

*****/

#include <stdio.h>

void main ()
{
    FILE * fic;      // handler du fichier texte à créer

    char nom [30];   // nom du fichier à créer

    printf ("** Creation et reaffichage d'un fichier texte **\n");

    // saisie du nom de fichier

```

```

printf ("Nom du fichier ? ");

scanf ("%30s", nom);

fflush (stdin);

// création du fichier texte

fic = fopen (nom, "w");

// saisie du contenu du fichier

printf ("Entrez un texte termine par une ligne vide :");
{
    char ligne [255];

    gets (ligne);    // lecture de la première ligne

    // tant que la ligne n'est pas vide
    while (strcmp (ligne, "") != 0)
    {
        // ajout du terminateur de ligne
        strcat (ligne, "\n");

        // écriture dans le fichier
        fputs (ligne, fic);

        // lecture de la ligne suivante
        gets (ligne);
    }
}

// fermeture du fichier

fclose (fic);
}

```

```

/*****

Programme    ficdirecte.c

Accès direct à un fichier binaire.

Ce programme stocke des structures de type "Descripteur" dans
un fichier binaire, puis relit une fiche de numéro choisi par $
l'utilisateur

*****/

#include <stdio.h>

#include <process.h>

// types décrits dans le fichier mesure.c

typedef char Libelle[20];
typedef enum { HS, OK } Etat;
typedef struct
{
    Libelle    lib;

    Etat    status;

    double    val;
} Descripteur;

// affichage complet de la structure des

void Afficher (char msg[], Descripteur des);

void main ()
{
    FILE * ficmesures ;    // handler du fichier

    // ouverture en lecture/écriture (w+) d'un fichier binaire (b)

    ficmesures = fopen ("mesures.dat", "w+b");

    if (ficmesures == NULL)

```

```

{
    printf ("Erreur ouverture ficmesures");
    exit(1);
}

// initialisation du fichier
{
    int i;
    int nb;

    Descripteur tampon;

    tampon.status = OK;

    for (i = 0; i < 5 ; i ++)
    {
        // initialisation de la structure à écrire dans le fichier
        sprintf ( tampon.lib,"PRES%d", i );

        tampon.val = 100.0 + i;

        // écriture d'un seul bloc (1) de la taille de la structure
        // Descripteur, dans le fichier ficmesures. Renvoie
        // le nombre d'enregistrement effectivement écrit (=> 1)
        nb = fwrite (&tampon, sizeof (Descripteur), 1, ficmesures);
        if (nb != 1)
        {
            printf ("Erreur ecriture fiche\n");
            exit (2);
        }
        else
            Afficher ("Ecriture :", tampon);
    }
}

```

```

// relecture d'une fiche
{
    int num;          // numéro de fiche à relire
    int retour ;      // code retour de la fonction fseek

    // choix du numéro de fiche à relire
    printf ("\n\nNumero de fiche a relire ?");
    scanf ("%d", &num);

    // Attention au calcul du déplacement :
    // pour accéder à la nième fiche, déplacement de
    // (n-1) * taille_fiche
    // SEEK_SET : déplacement à partir du début du fichier

    retour = fseek ( ficmesures,
                    (num-1) * sizeof (Descripteur),
                    SEEK_SET);

    // fseek renvoie 0 si OK
    if (retour !=0)
    {
        printf ("Erreur positionnement");
        exit (3);
    }

    // relecture de la fiche sélectionnée
    nb = fread ( &tampon, sizeof (Descripteur), 1,
                ficmesures);

    if (nb != 1)
    {

```



```

        printf ("Erreur lecture fiche\n");

        exit (4);

    }

    else

        Afficher ("\nRelecture :", tampon);

    }

}

// fermeture du fichier

fclose (ficmesures);

}

void Afficher (char msg[], Descripteur des)

{

    Libelle etats[2] = {"HS", "OK" };

    printf("\n%s\t", msg);

    printf ("Libelle:%s\t", des.lib);

    printf ("Etat      :%s\t", etats[des.status] );

    printf ("Valeur :%3.2f\n", des.val);

}

```



Exercice 9 : imprimer un état de nos mesures

En repartant de la maquette de l'exercice 8, on crée un fichier texte **Etat.txt** qui contient la description de tous les capteurs, à raison d'un capteur par ligne :

PRES0	Etat : OK	Valeur : 100.00
PRES1	Etat : HS	Valeur : 801.92
PRES2	Etat : OK	Valeur : 502.15
PRES3	Etat : OK	Valeur : 103.00
PRES4	Etat : HS	Valeur : 904.00

Une fois le fichier constitué, on l'imprimera par la fonction **system** qui permet d'appeler n'importe quelle commande du système d'exploitation :

```
system ("print etat.txt");
```



Exercice 10 : sauvegarder et recharger le tableau de structures dans un fichier binaire

Au début du programme, les mesures seront chargées du fichier binaire **Sauve.dat** s'il existe.

Sinon l'on partira d'un tableau vide : aucune mesure créée.

En fin de programme, les mesures modifiées seront recopiées dans le fichier binaire **Sauve.dat**.

S'il n'existe pas, il faudra le créer.

Pour réaliser cet exercice, on trouvera tous les éléments techniques dans l'exemple **ficdirecte.c**

Conseils de réalisation : ajouter un champ "fiche modifiée" à la structure Descripteur.

Ce champ sera mis à VRAI pour les nouvelles fiches ou les fiches modifiées, à FAUX pour les fiches qui n'ont pas changé depuis leur chargement du fichier de sauvegarde.

On ne recopiera à la fin que les fiches modifiées pendant la session de travail.