République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie d'Oran Mohamed BOUDIAF
USTO-MB

Algorithmique et Structures de Données II (ASD2)

Dr. Hafida Bouziane
Faculté des Mathématiques et Informatique
Département d'Informatique
e-mail: hafida.bouziane@univ-usto.dz

Á la mémoire de ma Chère mère

AVANT PROPOS

Cet ouvrage est une modeste contribution à l'enseignement de l'algorithmique, bagage de base pour l'art de la programmation. Comme on apprend par l'exemple et la pratique, ce document fournit des directives méthodologiques et des conseils à l'intention des novices résultant de plusieurs années d'enseignement de la matière. Il est consacré plus particulièrement à la maîtrise de la programmation modulaire, des fichiers et des structures de données linéaires. Il met l'accent sur quelques notions essentielles pour la conception des codes adéquats en tenant compte des avantages et inconvénients des structures de données et principes utilisés.

Ce document s'adresse aux étudiants du premier cycle universitaire suivant un enseignement d'informatique, plus particulièrement la première année Licence toutes spécialités confondues. Les algorithmes dans ce document sont rédigés dans un pseudo-code dont la syntaxe ressemble à celle du langage de programmation Pascal avec un rapprochement du C. Les exemples et les exercices proposés sont illustrés par des figures présentant les résultats de leur mise en œuvre sur machine.

Organisation du document

Le polycopié est réparti en trois chapitres :

Le premier chapitre est consacré aux sous-programmes et à la notion de récursivité. Il décrit les fonctions et les procédures et met l'accent sur les notions essentielles pour la mise en œuvre en C/C++ des programmes.

Le deuxième chapitre se focalise sur les fichiers et leur manipulation. Il décrit les principales fonctions permettant d'opérer sur les fichiers selon leur type (texte ou binaire) et le mode d'accès sollicité (séquentiel ou direct). Plusieurs exemples relatifs à chaque fonction introduite sont explicités avec détails.

Le troisième chapitre introduit la notion d'allocation dynamique de la mémoire qui est étroitement liée au type pointeur pour présenter les structures de données linéaires ou séquentielles : les listes, les piles

et les files. Il présente une description assez succincte du type abstraît de données mais plus explicite quant à son implémentation à travers ces structures de données de base. Les détails de chacune des implémentations sont présentés en tirant l'accent sur leurs avantages et inconvénients.

Conventions syntaxiques

Pour présenter les notions essentielles et les directives méthodologiques, certaines règles ont été adoptées, que nous résumerons dans ce qui suit :

- Les remarques importantes du cours sont mises en relief en utilisant des icônes indiquant leur nature.
- Les codes présentés dans cet ouvrage ont été implémentés à l'aide de l'environnement de développement Dev-C++.
- Ce qui est entre crochets « [ ] » est facultatif.
- Le symbole « | » veut dire ou bien.
- La présence de points de suspensions dans le code source, indique n'importe qu'elles autres instructions.
- Les constantes sont différenciées des variables par un mix entre majuscule et miniscule dans leurs identificateurs.

Remerciements

Je remercie tous ceux qui ont contribué de près ou de loin à l'élaboration de ce document et mes collègues de l'équipe d'Algorithmique qui ont accepté d'examiner le contenu et dont les avis ont été constructifs et encourageants pour une meilleure présentation de cette modeste contribution.

SOMMAIRE

CHAPITRE 1

LES SOUS-PROGRAMMES : FONCTIONS ET PROCÉDURES

Sommaire

Figures

1.1 Introduction

Un programme se complique et devient difficile à assimiler dés que le nombre d'instructions augmente, de plus certaines instructions ont tendance à se repéter plusieurs fois à différents endroits du programme. L'utilisation des sous-programmes (programmation modulaire) permet justement d'éviter cette redondance du code. Le principe consiste à détacher ces instructions qui se répètent du programme et de les reporter à part comme une portion de code indépendante identifiée par un nom, appelée sous-programme qui sera invoquée autant de fois qu'il est nécessaire, comme illustré dans la Figure 1.1.

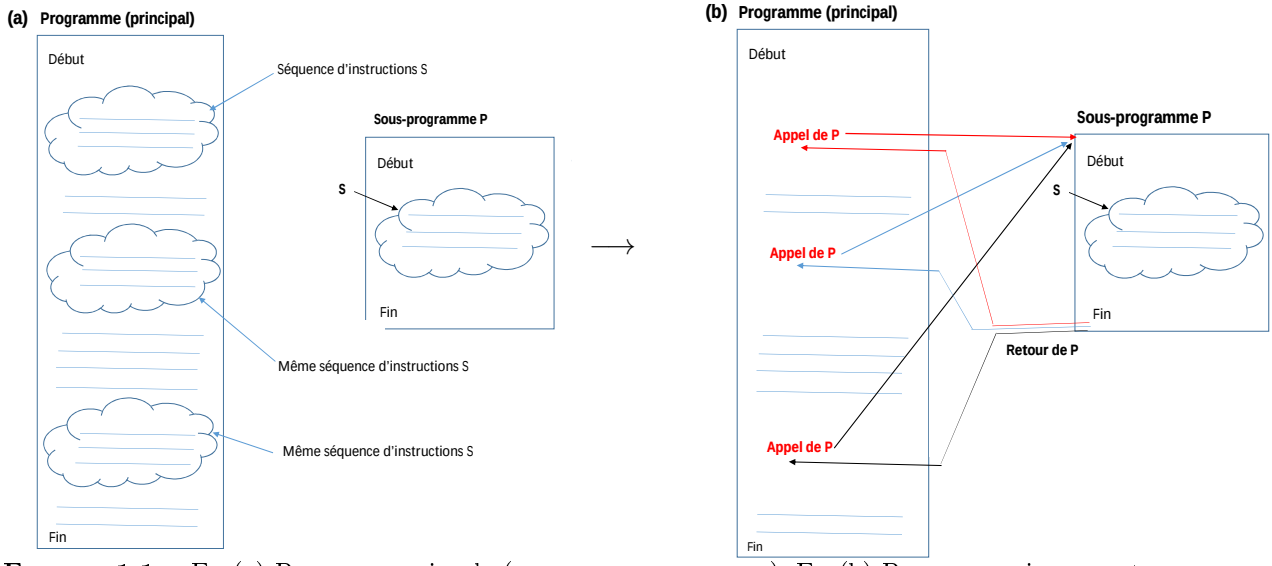


Figure 1.1 – En (a) Programme simple (sans sous-programmes). En (b) Programme invoquant un sous-programme.

Ainsi le programme devient plus simple, plus lisible et facile à maintenir. L'atout majeur de la programmation modulaire est la réutilisation du code.

1.2 Définitions

Un sous-programme a la même structure qu'un programme (principal), il contient donc une en-tête, une partie déclarative et un corps (instructions exécutables). Du point de vue algorithmique et en utilisant comme convenu la syntaxe du langage **Pascal**, le programme aura cette forme :

```
En-tête du programme principal
[Partie déclarative du programme principal]
En-tête du sous-programme
[Partie déclarative du sous-programme]
Début
Instructions du sous-programme
Fin;
Début
Instructions du programme principal
Fin.
```

Parmi les instructions du programme principal, on trouve l'appel au sous-programme définit précé–

demment. En langage Pascal, on distingue deux types de sous-programmes : la fonction et la procédure qui seront présentées plus en détail par la suite. Leurs en-têtes sont différentes puisque, la première retourne une valeur résultat et la seconde réalise un traitement sans retourner une valeur résultat mais en revanche, elle pourrait transmettre une ou plusieurs valeurs résultat d'une certaine manière qui sera explicitée dans la section 1.4. Leur définition obéit à la syntaxe suivante (ce qui est entre « (* » et « *) » est un commentaire) :

❀ Sous-programme de type **fonction**

```
fonction identificateur(arguments) : type; (* En-tête *)
[Partie déclarative de la fonction]
Début
Instructions de la fonction
identificateur = expression;
Fin;
```

Où **identificateur** est le nom de la fonction et **type** est le type de la valeur résultat retournée, obtenue en évaluant **expression** qui est exprimée en fonction des données communiquées par le programme principal à travers les arguments représentant le moyen de communication entre le programme et ses sous-programmes.

☞ Exemple :

```
fonction Max(a,b:entier) : entier;
Début
si a>b alors
      Max:=a;
      sinon
      Max:=b;
Fin;
```

Les arguments a et b sont communiqués par le programme principal, dans ce cas ce sont leurs valeurs qui sont transmises (nous verrons par la suite qu'il est aussi possible de transmettre leurs adresses en mémoire dans le cas où le programme appelant doit récupérer leurs nouvelles valeurs si elles sont modifiées dans le sous-programme). Max est le nom de la fonction qui contiendra la valeur résultat retournée au programme principal par la fonction.

❀ Sous-programme de type **procédure**

```
procédure identificateur(arguments); (* En-tête *)
[Partie déclarative de la procédure]
Début
Instructions de la procédure
Fin;
```

Où **identificateur** est le nom de la procédure qui contrairement à une fonction n'est pas destiné à contenir une valeur résultat, ce qui différencie la façon dont elle est invoquée au niveau du programme principal.

☞ Exemple :

```
procédure afficher(n:entier);
Var i : entiier;
Début
pour i := 1 à n faire
        écrire('*');
Fin;
```

La procédure effectue le traitement consistant à afficher n étoiles sur l'ecran sans retourner une valeur résultat.

☞ Exemple entier :

```
Algorithme Exemple;
VAR a,b,c,n:entier;
fonction Max(a,b:entier) : entier;
Début
si a>b alors
      Max:=a;
      sinon
      Max:=b;
Fin;
fonction afficher(n:entier);
Var i : entiier;
Début
pour i := 1 à n faire
        écrire('*');
Fin;
Début (* Programme principal *)
écrire('Entrer a et b :');
lire(a,b);
c := Max(a,b); (* Appel de la fonction Max *)
écrire('Le maximum est : ',c);
afficher(10); (* Appel de la procédure afficher *)
Fin.
```

Dans cet exemple, on remarque que l'appel de la fonction se trouve dans une partie droite d'une instruction d'affectation puisqu'elle retourne une valeur qui doit être stockée quelque part alors que l'invocation d'une procédure est une instruction à part entière. L'appel d'une fonction peut aussi se faire directement dans une instruction d'écriture pour afficher la valeur résultat retournée. En **C**, le programme principal **main()** est la fonction principale et les sous-programmes sont appelés fonctions secondaires. Elles comportent aussi une en-tête, une partie déclarative et un corps contenant les ins–tructions exécutables et sont placées avant ou après la fonction principale. Leur structure générale est comme suit :

```
Type identificateur (arguments)
{
[déclarations de variables internes]
 instructions;
 [return(expression);]
}
```

La première ligne représente l'en-tête de la fonction. Dans l'en-tête, **Type** désigne le type de la fonction, c'est-à-dire le type de la valeur du résultat qu'elle retourne. Une fonction peut éventuellement retourner un résultat qui est la valeur de **expression**. Dans le cas où le type spécifié est **void**, la fonction est équivalente à une procédure en **Pascal** ou une subroutine en **Fortran**, elle ne retourne pas de résultat par l'instruction **return** qui est omise mais nous verrons par la suite qu'elle pourra retourner un résultat ou plusieurs à travers ses arguments. Les arguments de la fonction sont appelés **paramètres formels**, ils obéissent à la syntaxe suivante :

Type$_1$ **identificateur**$_1$, **Type**$_2$ **identificateur**$_2$,..., **Type**$_n$ **identificateur**$_n$

Où chaque paramètre indiqué par son identificateur est précédé par son type respectif. Les paramètres formels peuvent être de n'importe quel type (int, float, double, char,...etc.). Leurs identificateurs n'ont d'importance ou de signification qu'à l'intérieur de la fonction. Si la fonction ne possède pas d'arguments, ceux ci sont remplacés par le mot-clé **void** dans l'en-tête ou tout simplement le vide. Plusieurs instructions **return** peuvent apparaître dans le corp d'une fonction à différents endroits. Le retour à la fonction appelante aura lieu dès la rencontre du premier **return** lors de l'exécution de la fonction.

☺ Dans le cas où la fonction secondaire est définie après la fonction principale **main()**, il est indispensable qu'elle soit déclarée avant que le compilateur rencontre un appel à celle ci. La notion de **prototype** permet de déclarer au préalable les fonctions secondaires placées après la fonction **main()**, en spécifiant le type de la fonction et celui de ses paramètres comme suit :

Type identificateur(Type$_1$**,Type**$_2$**,...,Type**$_n$**) ;**

Ou bien :

Type identificateur(Type$_1$ **identificateur**$_1$**,Type**$_2$ **identificateur**$_2$**,**
...,Type$_n$ **identificateur**$_n$**) ;**

Attention

☺ Le prototype d'une fonction secondaire se termine par un point virgule « ; » contrairement à l'en-tête qui n'est pas considérée comme une instruction.

L'appel d'une fonction secondaire se fait par l'instruction :

identificateur(identificateur$_1$**,identificateur**$_2$**,...,identificateur**$_n$**)**

Où identificateur$_1$, identificateur$_2$,..., identificateur$_n$ sont **les paramètres effectifs ou réels**. Les paramètres formels et effectifs doivent s'accorder en nombre, ordre et leurs types doivent être compatibles. Contrairement aux paramètres formels, les paramètres effectifs peuvent être également des expressions qui seront évaluées lors de l'appel de la fonction.

☺ Le paramètre formel n'a pas forcément le même nom que le paramètre effectif.

Déconseillé

☹ Les opérateurs d'incrémentation (++) et de décrémentation (- -) sont à éviter dans les expressions définissant les paramètres effectifs.

Il est à noter que la forme de l'appel diffère pour une fonction retournant un résultat par **return** et une fonction **void** (procédure), on distingue ces deux formes :

- Fonction avec return :

  variable = identificateur(identificateur$_1$,identificateur$_2$,...,identificateur$_n$) ;

  Où **variable** représente le nom de la case mémoire destinée à contenir le résultat retourné par la fonction appelée. Donc l'appel de la fonction se trouve dans **une partie droite** d'une affectation contenant l'appel. L'appel à la fonction peut aussi être placé dans une instruction d'écriture directement sans avoir recours à une case mémoire pour le stocker comme il est montré dans Exemple1 et Exemple2.

- Fonction void (procédure) :

  identificateur(identificateur$_1$,identificateur$_2$,...,identificateur$_n$) ;

  L'invocation de la fonction se fait par le nom suivi par les paramètres effectifs, sous forme d'une instruction à elle seule comme il est montré dans Exemple3.

☞ Exemple1 : fonctions secondaires placées avant la fonction principale.

```
#include <stdio.h>
//Codes des deux fonctions secondaires
float perimetre(float longueur,float largeur) /* longueur et largeur sont les paramètres formels */
{
return 2*(longueur+largeur);
}
float surface(float longueur,float largeur)
{
return longueur*largeur;
}
//Code de la fonction principale (appelante)
int main(){
float longueur,largeur;
printf("Entrer la longueur et la largeur du rectangle : ");
scanf("%f%f",&longueur,&largeur);
//Appels des deux fonctions secondaires: 2 en 1 affichage et appel
printf("Le perimetre est : %f\nLa surface est : %f\n", perimetre(longueur,largeur),surface(longueur,
    largeur)); /* longueur et largeur sont les paramètres effectifs (réels) */
}
```

Exemple d'exécution du code :

```
Entrer la longueur et la largeur du rectangle : 12.5
5.5
Le perimetre est : 36.000000
La surface est : 68.750000

--------------------------------
Process exited after 225.8 seconds with return value 0
Appuyez sur une touche pour continuer...
```

Figure 1.2 – Exemple d'exécution du code calculant le périmètre et la surface d'un rectangle.

Les fonctions secondaires peuvent être placées après la fonction principale.
☞ Exemple2 : fonctions secondaires placées après la fonction principale.

```
#include <stdio.h>
//Prototypes des deux fonctions secondaires
float perimetre(float,float);
float surface(float,float);
//Code de la fonction principale (appelante)
int main(){
float longueur,largeur;
printf("Entrer la longueur et la largeur du rectangle : ");
scanf("%f%f",&longueur,&largeur);
//Appels des deux fonctions secondaires
printf("Le perimetre est : %f\nLa surface est : %f\n", perimetre(longueur,largeur),surface(longueur,
    largeur));
}
//Codes des deux fonctions secondaires
float perimetre(float longueur,float largeur)
{
return 2*(longueur+largeur);
}
float surface(float longueur,float largeur)
{
return longueur*largeur;
}
```

☞ Exemple3 : fonction void (procédure).

```
#include <stdio.h>
void saluer() // Ou bien void saluer(void)
{
printf("Salut tout le monde !\n");
}
int main(){
// Juste un salut
saluer(); // La fonction n'a pas de paramètres
}
```

1.3 Portée des données

La portée d'une variable est l'ensemble des fonctions où cette variable est connue. Les instructions de ces fonctions peuvent alors en disposer. Selon ce concept, on distingue deux sortes de variables :

- Une variable définie en dehors de toute fonction, y compris la fonction principale **main()**, elle est dite **globale** car toute fonction définie dans le programme peut en disposer. Cette variable est permanente (statique) puisqu'elle dure jusqu'à la fin de l'exécution du programme.
- Une variable définie au sein d'une fonction est **locale** à cette fonction et sa portée est limitée uniquement à cette fonction, elle est dynamique puisque sa durée de vie est limitée, elle dure le temps que la fonction en question s'exécute et expire (sa valeur est perdue) à la fin de l'exécution de celle-ci.

Les variables locales n'ont aucun lien avec les variables globales de même nom, c'est à dire lorsqu'une variable locale et une variable globale sont identifiées par un même nom, la variable globale est localement masquée (inaccessible), ce qui veut dire que c'est la variable locale qui est utilisée.
☞ Exemple :

```
# include <stdio.h>
int i=1, j, k ; // Variables globales
float x; // Variable globale
void fonction() {
float x=1, y ; // Variables locales à fonction()
i++ ; // La variable globale i est modifiée
x*= 10 ; // C'est uniquement la variable locale x qui est modifiée
y = x/2 ;
}
int main() {
j=2 ; k=i+3 ;
fonction() ;
printf("\ni=%d j=%d  k=%d  x=%.2f\n",i,j,k,x);
}
```

Résultat du code en question :

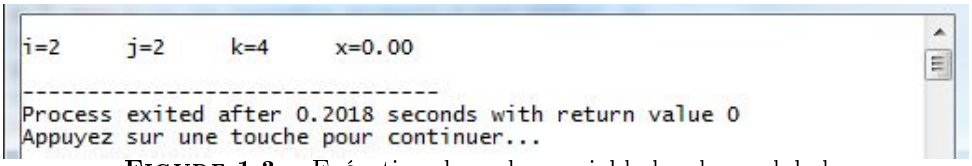


Figure 1.3 – Exécution du code : variable locale vs globale.

☺ Cette notion de portée est aussi valable pour les constantes et les types.

1.4 Passage des paramètres

Les fonctions d'un programme communiquent à travers les paramètres effectifs et les paramètres formels. Leur mise en relation peut avoir comme elle peut ne pas avoir de conséquences sur les valeurs initiales des paramétres effectifs, tout dépend donc de la manière de transmission des paramètres effectifs.

1.4.1 Passage par valeur et par adresse

Le passage des valeurs des paramètres effectifs aux paramètres formels peut se faire de deux façons :

Passage par valeur : au moment de l'appel de la fonction, une copie du paramètre effectif est affectée au paramètre formel mais cette affectation n'a pas d'incidence sur le paramètre effectif. La valeur initiale du paramètre effectif reste inchangée après l'exécution (le retour) de la fonction.
☞ Exemple :

```
Algorithme Exemple;
VAR x,s; (* Partie déclarative du programme principal *)
fonction carre(x:réel):réel; (* Passage par valeur de x *)
Début
   carre :=  x*x;
Fin;
procédure calculer(s:réel,n:entier) (* s et n passés par valeur*)
Début
s = s+carre(x)/n; (* s est modifié dans la procédure *)
écrire('Dans la fonction s= ',s);
Fin;
Début (* Programme principal *)
x:=5; s:=0;
calculer(s,10); (* Appel de la procédure calculer *)
écrire('Dans le programme principal s=',s);
Fin.
```

En C :

```
#include<stdio.h>
float x=5, s=0; // variables globales
float carre(float x){
   return x * x;}
void calculer(float s,int n){
s+=carre(x)/n;
printf("Dans la fonction s=%f\n",s);}
int main(){
calculer(s,10);
printf("Dans le main s=%f\n",s);}
```

Lors de l'appel de la fonction **calculer**, la valeur de n est 10 et s vaut 0. Bien que la valeur de s ait été modifiée dans la procédure **calculer**, sa valeur dans le main reste intacte comme le montre le résutat :

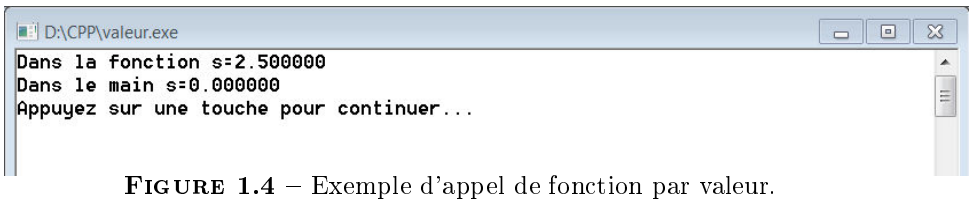


Figure 1.4 – Exemple d'appel de fonction par valeur.

Passage par variable ou référence : Ce mode est utilisé dans le cas où la fonction appelante doit récupérer le changement qu'a opéré la fonction appelée sur le paramètre effectif. Au moment de l'appel de la fonction, le paramètre effectif et le paramètre formel font référence à la même case mémoire puisque la fonction appelante communique à la fonction appelée l'adresse du paramètre effectif. Ainsi toute modification de la valeur du paramètre formel entraînera automatiquement la modification de la valeur du paramètre effectif puisque la fonction a accès à l'emplacement mémoire du paramètre. Dans l'algorithme et par analogie au langage Pascal, dans le cas d'un passage par adresse (référence) le paramètre formel est précédé par le mot clé

VAR. La permutation de deux variable x et y est un bon exemple pour comprendre ce mode de transmission des paramètres, soit l'algorithme suivant :

```
Algorithme Exemple;
VAR x,y: réel;
fonction echanger(VAR x:réel; VAR y:réel) (* Appel par adresse pour x et y *)
VAR z:réel;
Début
z:=x;
x:=y;
y:=z;
Fin;
Début (* Programme principal *)
écrire('Entrer les valeurs de x et y : ');
lire(x,y);
écrire('Avant permutation :');
écrire('x=',x,'y=',y);
echanger(x,y); (* Appel par adresse *)
écrire('Après permutation :');
écrire('x=',x,'y=',y);
Fin.
```

Dans l'en-tête de la procédure, les deux variables x et y sont précédées par le mot clé **VAR** pour indiquer le passage des paramètres par adresse (référence) afin que le programme appelant puisse récupérer les deux variables modifiées par la procédure. Le langage C n'utilise pas un mot clé pour marquer ce mode de transmission mais plutôt la notion de pointeur comme suit :
Code en C :

```
#include <stdio.h>
void echanger(float *x, float *y){ // Appel par adresse
float z;
z=*x;
*x=*y;
*y=z;}
int main(){
    float x,y;
    printf("Entrer les valeurs de x et y : ");
    scanf("%f%f",&x,&y);
    printf("Avant permutation : \n");
    printf("x= %f       y=%f\n",x,y);
    echanger(&x,&y); // Appel par adresse
    printf("Apres permutation : \n");
    printf("x= %f       y=%f\n",x,y);}
```

☺ Le langage C++ a simplifié bien des contraintes en C comme par exemple pour l'appel de fonctions par adresse, l'opérateur (*) est omis dans le corps des fonction, seul l'opérateur (&) est mentionné dans les paramètres formels pour indiquer l'appel par adresse. Au niveau des paramètres effectifs l'appel se fait comme l'appel par valeur, c'est à dire l'opérateur & est omis.

Code en C++ :

```
#include <stdio.h>
void echanger(float &x, float &y){ // Appel par adresse
float z;
z=x;
x=y;
y=z;}
int main(){
    float x,y;
    printf("Entrer les valeurs de x et y : ");
    scanf("%f%f",&x,&y);
    printf("Avant permutation : \n");
    printf("x= %f       y=%f\n",x,y);
    echanger(x,y);
    printf("Apres permutation : \n");
    printf("x= %f       y=%f\n",x,y);}
```

Résultat de l'exécution :

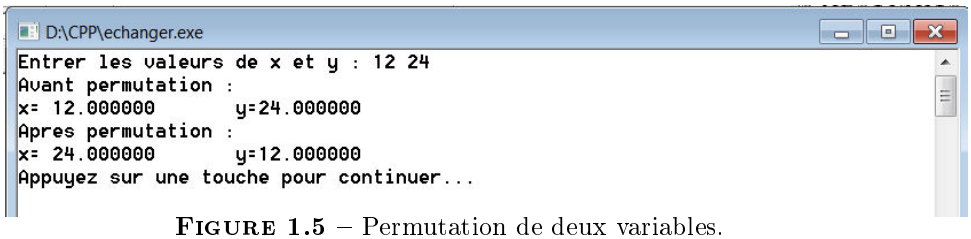


FIGURE 1.5 – Permutation de deux variables.

Il est aussi possible de profiter de la notion de variable globale pour récupérer les modifications opérées sur les variables au sein des fonctions sans avoir recours à l'appel par adresses sur les variables, comme illustré dans le code ci-dessous. Dans ce cas, les modifications sont alors implicites.

```
#include <stdio.h>
float x,y; /* Les variables x et y sont globales (leurs cases mémoires sont accessibles par toutes les
     fonctions) */
void echanger(){ // Pas besoin d'envoyer les adresses de x et y  puisqu'elles sont globales
float z;
```

```
z=x;
x=y;
y=z;}
int main(){
    printf("Entrer les valeurs de x et y : ");
    scanf("%f%f",&x,&y);
    printf("Avant permutation : \n");
    printf("x= %f       y=%f\n",x,y);
    echanger();  //  Fonction sans paramètres puisque x et y sont globales
    printf("Apres permutation : \n");
    printf("x= %f       y=%f\n",x,y);}
```

1.4.2 Tableaux comme paramètres d'une fonction

Un tableau est en fait un pointeur sur une zone mémoire où sont stockées toutes les cases occupées par ses éléments de façon consécutive. Bien qu'il est question à ce niveau de la notion de pointeur, il convient à priori de retenir qu'un pointeur est tout simplement une adresse mémoire. La notion de pointeur est détaillée dans le Chapitre 3, Section 3.2 pour faciliter la compréhension des listes chaînées.

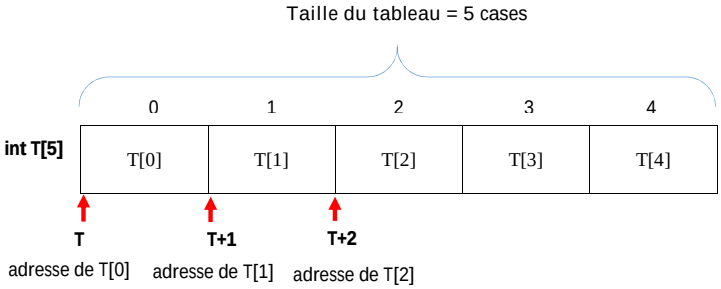


Figure 1.6 – Le nom du tableau pointe vers sa première case (son premier élément).

T est un pointeur sur la première case du tableau, les expressions $T[0]$ et *T désignent le contenu de la première case (premier élément) du tableau. La valeur $T+1$ pointe sur la deuxième case (second élément) du tableau et $T[1] = *(T+1)$ désigne le contenu de la deuxième case du tableau (second élémment) et ainsi de suite.
Il est possible d'obtenir l'adresse d'une case du tableau à l'aide de l'opérateur & comme suit :

T+i = & T[i]
et
T[i] = *(T+i)

Un nom de tableau est en fait l'adresse de la zone mémoire où est stocké le contenu du tableau. Par conséquent, passer un tableau en paramètre à une fonction secondaire revient à passer son adresse, c'est donc un appel de la fonction par adresse ou référence implicite sur le paramètre tableau, il n'est pas nécessaire d'utiliser l'opérateur & (en C) pour le préciser. Toute modification du paramètre effectif du type tableau entraîne par défaut une modification du paramètre formel correspondant.
☞ Exemple1 : afficher le minimum et le maximum se trouvant dans un tableau d'entiers (appel par

valeur sur le tableau puisqu'il n'est pas modifié). Pour simplifier le code, l'appel par adresses des variables est fait selon le C++.

```
#include <stdio.h>
#define TAILLE 50
void minmax(int V[TAILLE], int n, int &min, int &max, int &posmin, int &posmax);
main(void)
{
int V[TAILLE], i, n, posmin, posmax, min, max;
printf("donner la taille n :");
  scanf("%d", &n);
  printf("donner le tableau element par element :");
  for(i=0;i<n;i++)
  scanf("%d", &V[i]);
  minmax(V, n, min, max, posmin, posmax);
  printf("Le min est : %d   et sa position est : %d\n", min, posmin+1);
  printf("Le max est : %d   et sa position est : %d\n", max, posmax+1);
}
void minmax(int V[TAILLE], int n, int &min, int &max, int &posmin, int &posmax)
{int i; / * i est locale à la fonction */
min = V[0]; posmin = 0;
max = V[0]; posmax = 0;
for(i=1;i<n;i++)
 {if(V[i]<min)
   {
    min = V[i];
    posmin = i;
   }
  if(V[i]>max)
   {
    max = V[i];
    posmax = i;
    }
   }
}
```

Résultat de l'exécution :

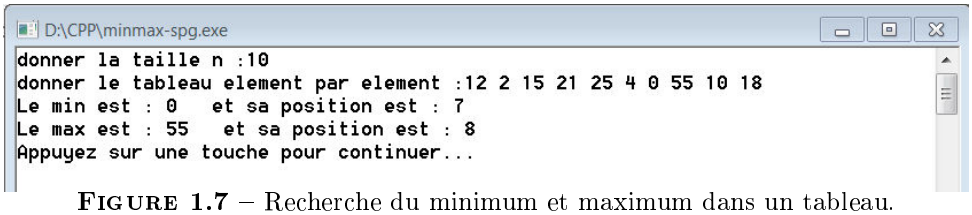


Figure 1.7 – Recherche du minimum et maximum dans un tableau.

Observer le code suivant du même exemple écrit autrement et dont le résultat est exactement le même :

```
#include <stdio.h>
#define TAILLE 50
int V[TAILLE], i, n, posmin, posmax, min, max; // Déclaration globale des variables
void minmax();
main(void){
printf("donner la taille n :");
  scanf("%d", &n);
  printf("donner le tableau element par element :");
  for(i=0;i<n;i++)
  scanf("%d", &V[i]);
  minmax();
  printf("Le min est : %d   et sa position est : %d\n", min, posmin+1);
  printf("Le max est : %d   et sa position est : %d\n", max, posmax+1);
}
void minmax(){
min = V[0]; posmin = 0;
max = V[0]; posmax = 0;
for(i=1;i<n;i++)
 {if(V[i]<min)
   {
    min = V[i];
    posmin = i;
   }
  if(V[i]>max)
   {
    max = V[i];
    posmax = i;
    }
    }
   }
```

Résultat de l'exécution :

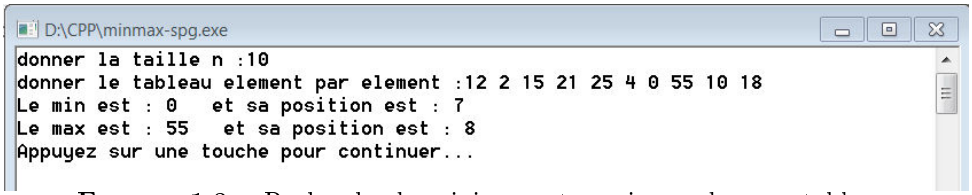


FIGURE 1.8 – Recherche du minimum et maximum dans un tableau.

Aucun paramètre n'est transmis à la fonction **minmax()** puisque toutes les données qu'elle utilise sont déclarées globales, ainsi toute modification dans la fonction des données entraîne automatiquement la modification des mêmes données dans la fonction principale puisque la fonction a accès à leur emplacement mémoire.

☞ Exemple2 : insérer une valeur VAL dans un tableau trié d'entiers (appel par référence sur le tableau pour récupérer le tableau modifié). Le code est en C pour mieux faire apparaître l'appel par adresse ou référence.

```c
#include <stdio.h>
#define TAILLE 50
void insert(int T[TAILLE],int *n,int VAL)
{ int i;   /* i est locale à la fonction */
 /* Déplacer les éléments plus grands que
    VAL d'une position vers l'arrière  */
 for (i=*n ; (i>0&&T[i-1]>VAL) ; i--) /* Décaler tous les éléments > VAl vers l'arrière */
    T[i]=T[i-1];
 T[i]=VAL; // L'élément à la potion i est remplacé par VAL
/* Nouvelle dimension du tableau */
 *n=*n+1;
}
int main(){
 int T[TAILLE],n,i,VAL;     /* i indice local au main    */
 /* Saisie des données */
 printf("Dimension n du tableau initial (Taille max 50) : ");
 scanf("%d", &n );
 for (i=0; i<n; i++)
 {printf("Element T[%d] : ", i);
  scanf("%d", &T[i]);}
 printf("Valeur a inserer : ");
 scanf("%d", &VAL );
 /* Affichage du tableau initial */
 printf("Tableau initial : \n");
 for (i=0; i<n; i++)
    printf("%d ", T[i]);
 insert(T,&n,VAL); // Passage par adresse implicite du tableau T
 /* Affichage des résultats */
 printf("\nTaille du nouveau tableau = %d \n",n);
 printf("Tableau final :\n");
 for (i=0; i<n; i++)
    printf("%d ", T[i]);
}
```

Résultat de l'exécution :

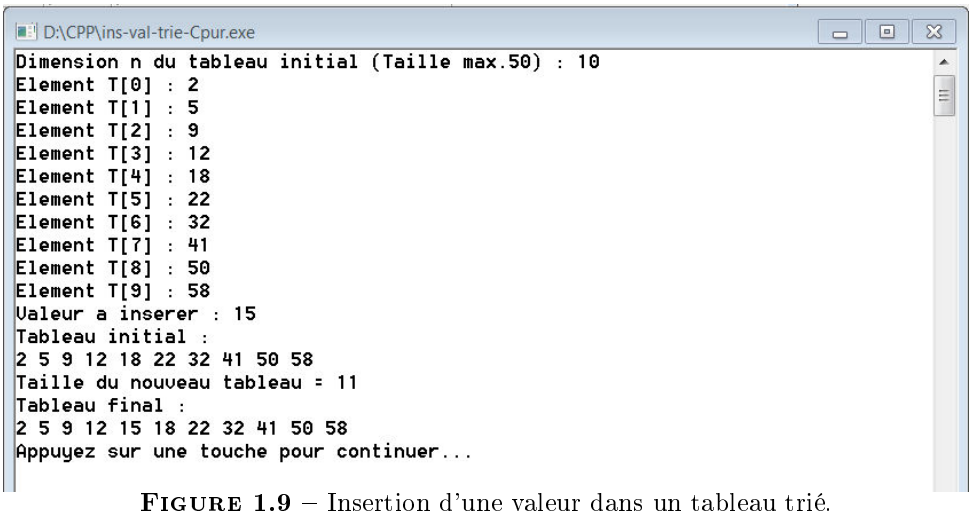


Figure 1.9 – Insertion d'une valeur dans un tableau trié.

☺ La programmation modulaire permet une écriture plus claire du code et a de nombreux atouts.
☺ Un sous-programme peut en appeler un autre comme il peut s'appeler lui même, c'est la notion de récursivité présentée ci-dessous.

1.5 La récursivité

1.5.1 Définitions

Un algorithme est dit récursif s'il est défini en fonction de lui même  (Quercia, 2002). La notion de récursivité est étroitement liée à la notion de récurrence mathématique qui présente une forme de répétition  (Berthet and Labatut, 2014a). Un sous-programme (ou fonction secondaire) est dit récursif s'il s'appelle lui même. L'appel peut être direct ou explicite (récursivité directe) ou implicite quand le sous-programme appelle un autre sous-programme et celui ci lui fait appel (récursivité indirecte). Si le sous-programme s'appelle une fois dans lui même, c'est une récusivité simple dite aussi d'ordre un. Si par contre, il fait plusieurs appels à lui même, la récursivité est d'ordre multiple.
Principe :
La récursivité transforme un processus itératif (boucle) en une série d'appels de sous-programme qui est repétée jusqu'à la rencontre d'une condition d'arrêt (comme dans le cas d'une boucle) communément dite cas particulier ou de base où la taille du problème à résoudre devient très petite voire irréductible menant à une solution immédiate, autrement les appels se succèdent réduisant la taille du problème. Etant donné un problème, écrire un algorithme récursif pour le résoudre consiste donc à le découper en sous-problèmes de même nature de taille réduite, les résoudre puis à partir des solutions partielles, résoudre le problème initial, ce qui consiste à :
- identifer le ou les cas de base.
- identifier le cas général marquant la récursivité.
Forme générale :

```
En-tête du sous-programme
[Partie déclarative du sous-programme]
Début
si(condition) alors (* Il peut y avoir plusieurs *)
      instruction du cas de base  (* Solution immédiate *)
       sinon
       Début
       [instruction] (* Il peut y avoir plusieurs *)
       Appel au sous-programme (* Réduire la taille du problème *)
       [instruction] (* Il peut y avoir plusieurs *)
       Fin;
Fin;
```

Le bloc d'instructions suivant le **sinon** peut être à la limite une seule instruction, celle de l'appel au sous-programme. La taille du problème initial apparaît au niveau du premier appel du sous-programme à travers les paramètres effectifs. Le mécanisme de la récursivité réduit progressivement cette taille qui

aboutit à un cas de base.

☞ Exemple : soit une fonction récursive(int d) résolvant un problème Pb de taille d de sorte que :

$$recursive(d) = \begin{cases} 1 & si \;\; d \leq 0 \;\; \rightsquigarrow \;\; cas \;\; de \;\; base \\ d + recursive(d-1) & si \;\; d > 0 \;\; \rightsquigarrow \;\; cas \;\; général \end{cases} \tag{1.1}$$

```
Algorithme Exemple;
VAR
   d: entier;     (* Taille du problème *)
fonction recursive(d:entier) : entier;
Début
si d<=0 alors
      recursive := 1;     (* Cas de base *)
      sinon
      recursive := d + recursive(d-1); (* Cas général *)
Fin;
Début (* Programme principal *)
écrire('Entrer la taille du problème Pb :');
lire(d);
écrire('Solution de Pb = ', recursive(d)); (* Premier appel  à la fonction recursive *)
Fin.
```

Supposons que Pb est de taille d=4, en déroulant la fonction récursive pour d=4, nous aurons le schéma illustré dans la Figure 1.10.

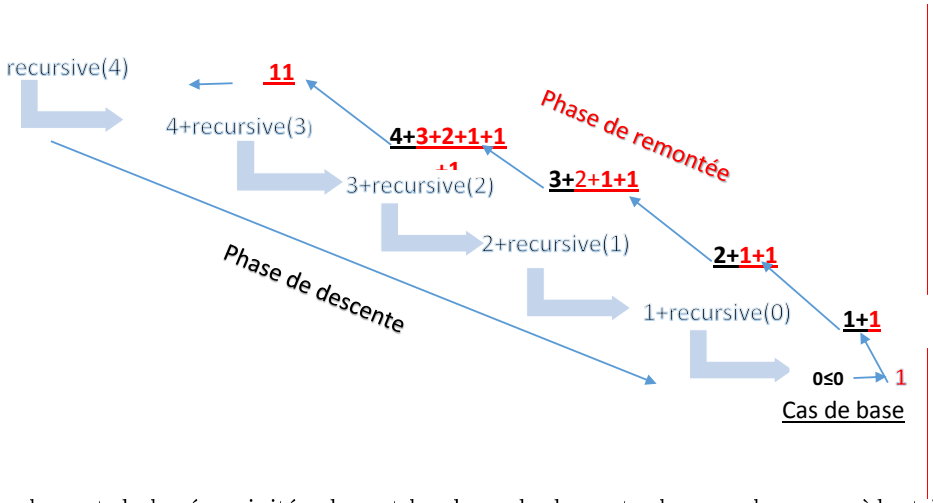


Figure 1.10 – Déroulement de la récursivité : durant la phase de descente, les appels se succèdent jusqu'à la rencontre du cas de base qui déclenche la phase de remontée (produisant des solutions partielles) qui évolue jusqu'au premier appel récursif qui termine le processus, produisant la solution finale du problème en entier.

1.5.2 Types de récursivité

❀ Récursivité simple :

Le sous-programme/fonction secondaire contient un appel à lui même.

☞ Exemple1 : la fonction puissance x^n.

$$x^n = \begin{cases} 1 & si \;\; n = 0 \;\; \rightsquigarrow \;\; cas \;\; de \;\; base \\ x \cdot x^{n-1} & si \;\; n \geq 1 \;\; \rightsquigarrow \;\; cas \;\; général \end{cases} \tag{1.2}$$

Version itérative :

```
float puis(float x, int n){
float p=1;
  for(i=1;i<=n;i++)
      p = p*x;
      return p;
}
```

Version récursive :

```
float puis(float x, int n){
if(n==0)   // Cas de base
      return 1 ;
      else
      return x*puis(x,n-1);  // Cas général
}
```

☞ Exemple2 : la factorielle d'un nombre entier $n \geq 1$, n != $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 = \prod_{i=1}^{n} i$.

$$n! = \begin{cases} 1 & si \quad n \leq 1 \\ n \cdot (n-1)! & sinon \end{cases} \tag{1.3}$$

Version itérative :

```
unsigned long fact(int n){
int i;
unsigned long f=1;
for(i=1;i<=n;i++)
      f=f*i;
return f;
}
```

Version récursive :

```
unsigned long fact(int n){
if(n==0)  // Cas de base
      return 1 ;
      else
      return n*fact(n-1); // Cas général
}
```

☞ Exemple3 : le calcul du Plus Grand Commun Diviseur (PGCD) de deux nombres entiers a et b.

L'algorithme d'Euclide ou des divisions successives : d'après l'algorithme d'Euclide ($\tilde{3}00$ avant J.C.), le plus grand commun diviseur de 2 nombres entiers a et b est calculé en appliquant la relation de récurrence :

$pgcd(a, b) = pgcd(b, a\%b)$ si $b \neq 0$ jusqu'à ce que le reste $(a\%b)$ soit nul.

```
int pgcd(int a,int b){
int r;
if(a<b)
  return pgcd(b,a);
do {
   r=a%b;
   a=b;
   b=r;
  } while(r!=0);
return a;
}
```

```
int pgcd(int a,int b){
if(a<b)
     return pgcd(b,a);
if(b==0)
   return a;
   else
   return pgcd(b,a%b);
}
```

Méthode des différences successives : calculer le PGCD de deux nombres revient à calculer le PGCD de la différence des deux nombres et du plus petit d'entre eux jusqu'à l'obtention d'une différence nulle.

Version itérative :

```
int pgcd(int a, int b){
 do
 if(a>b)
 a=a-b;
 else
 b=b-a;
 while(a!=b);
 return a;
 }
```

Version récursive :

```
int pgcd(int a, int b){
if(a==b)
  return a ;
  else
  if(a>b)
    return pgcd(a-b,b);
    else
    return pgcd(a,b-a);
}
```

☞ Exemple4 : fonction vérifiant si un mot lu en entrée est palindrome, c'est à dire qu'il soit lu de droite à gauche ou de gauche à droite, il restera le même comme par exemple : ici, elle, kayak, radar, LOL, etc.

Version itérative :

```
int palindrome(char mot[]){
 int i,n;
 n = strlen(mot);
 for(i=0;i<n/2;i++)
        if(mot[i] != mot[n-i-1])
                    return 0; // Le mot n'est pas palindrome
return 1; // Le mot est palindrome
}
```

Version récursive : l'appel au niveau du main se fait par **palindrome(mot,0,strlen(mot)-1)** aprés avoir inclus la librairie <string.h> pour utiliser la fonction strlen(mot) qui retourne la longueur du mot et donc initialement $deb = 0$ et $fin = strlen(mot) - 1$.

```
int palindrome(char mot[], int deb, int fin){
 if(deb>fin) // On finit par être en dehors du tableau
   return 1; // le mot est palindrome
   else
   if(mot[deb] == mot[fin]) /* Si le caractère de gauche est identique au caractère de droite */
       return palindrome(mot,deb+1,fin-1); /* Tester les autres caractères restants */
       else
       return 0;    // Autrement le mot n'est pas palindrome
}
```

La Figure 1.11 donne un exemple d'exécution de la fonction palindrome.

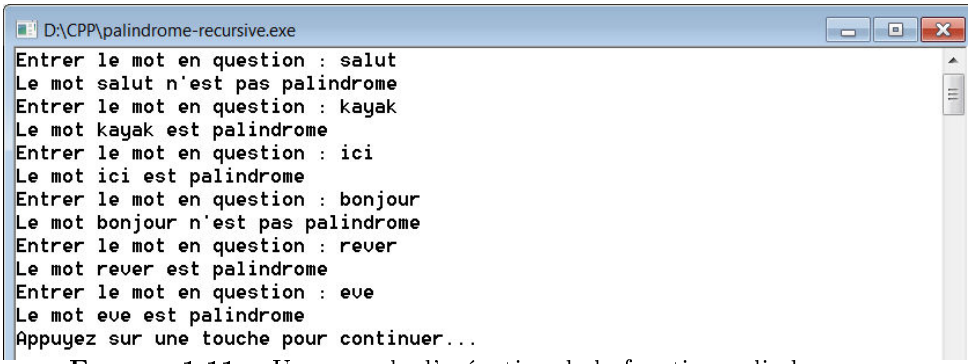


FIGURE 1.11 – Un exemple d'exécution de la fonction palindrome.

❀ Récursivité multiple : Le sous-programme/fonction secondaire fait appel à lui même plusieurs fois au même temps.

☞ Exemple1 : les nombres de Fibonacci (mathématicien italien Léonard de Pise surnommé Fibonacci, début du XIII-ème siècle) (Posamentier and Lehmann, 2007).

$$F_n = \begin{cases} 1 & si \;\; n \leq 1 \\ F_{n-1} + F_{n-2} & sinon \;\; \rightsquigarrow \;\; récursivité \;\; d'ordre \;\; 2 \end{cases} \tag{1.4}$$

```
int fib(int n){
if(n<=1)
      return 1 ;
      else
      return fib(n-1)+fib(n-2); // Récursivité d'ordre 2
}
```

On peut observer que le calcul de fib(5) par exemple produit 15 appels à la fonction comme il est illustré dans la figure 1.12

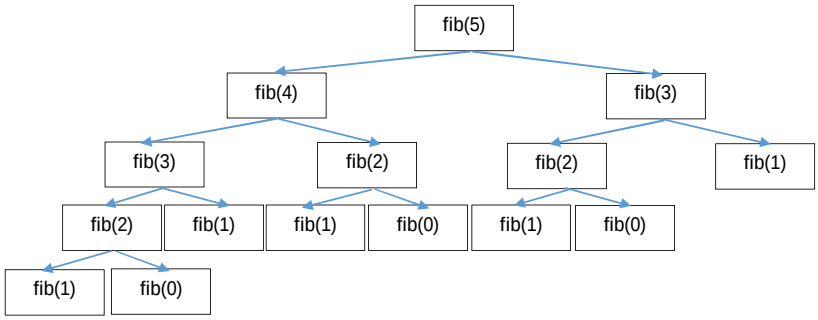


FIGURE 1.12 – Appels récursifs pour calculer fib(5)

☹ Dans le cas de l'application de la récursivité pour calculer les nombres de Fibonacci, on remarque que le nombre d'appels à la fonction **fib** augmente de façon exponentielle. Comme l'appel à une fonction secondaire est plus coûteux qu'une simple opération arithmétique, ceci implique un temps d'exécution important. Dans ce cas, la version itérative en utilisant des variables simples est beaucoup plus intéressante vu son calcul à moindre coût par rapport à la version récursive.

```
int fib(int n){
int i,f0=1,f1=1,f=1;
for(i=1;i<=n;i++)
   {
   f=f0+f1;
   f0=f1;
   f1=f;
   }
return f;
}
```

☞ Exemple2 : le problème des tours de Hanoi imaginé par le mathématicien français Edouard Lucas, fin du XIX-siècle (Lucas, 1892). On dispose de trois tours (départ, arrivée ou destination et intermédiaire ou temporaire) et de 64 disques, tous de rayons différents comme le montre la

Figure 1.13.

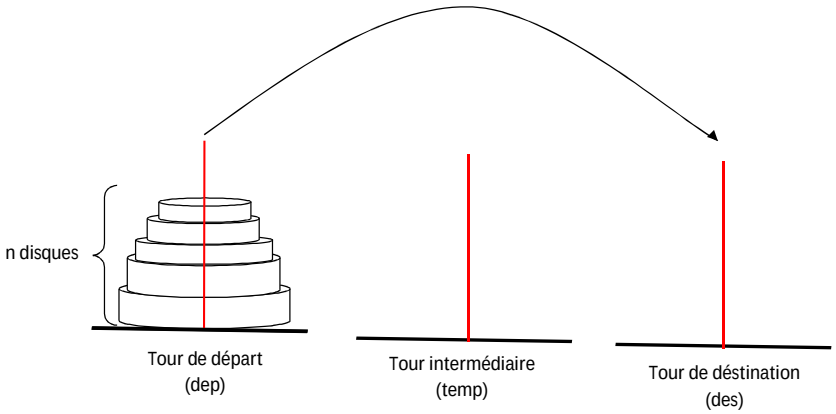


FIGURE 1.13 – Problème des tours de Hanoi

Au départ les 64 disques sont au niveau de la tour de départ, rangés par taille, le plus grand tout en bas. Le but est de déplacer ces disques pour les amener sur la troisième tour en suivant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois ;
- à chaque étape et sur chaque tour, un disque ne peut être placé qu'au-dessus d'un disque de rayon plus grand ou le vide.

En représentant les tours par exemple par des nombres entiers (ou caractères), où dep =1, des=3 et temp=2, l'appel de la procédure au niveau du main se fait par : hanoi(64,1,3,2) ;
Il a été démontré que si n est le nombre de disques, il faut au moins $2^n - 1$ étapes pour y arriver. Pour déplacer n disques de la tour de départ vers la tour de destination, il va falloir effectuer trois opérations consécutives :

- Déplacer les $(n-1)$ disques de dep vers temp ($(n-1)$ déplacements).
- Déplacer le n^{me} disque (le plus grand disque restant) de dep vers des.
- Déplacer les $(n-1)$ disques de temp vers des ($(n-1)$ déplacements).

Prenons le cas de n=3 disques, la Figure 1.14 détaille les étapes à effectuer.

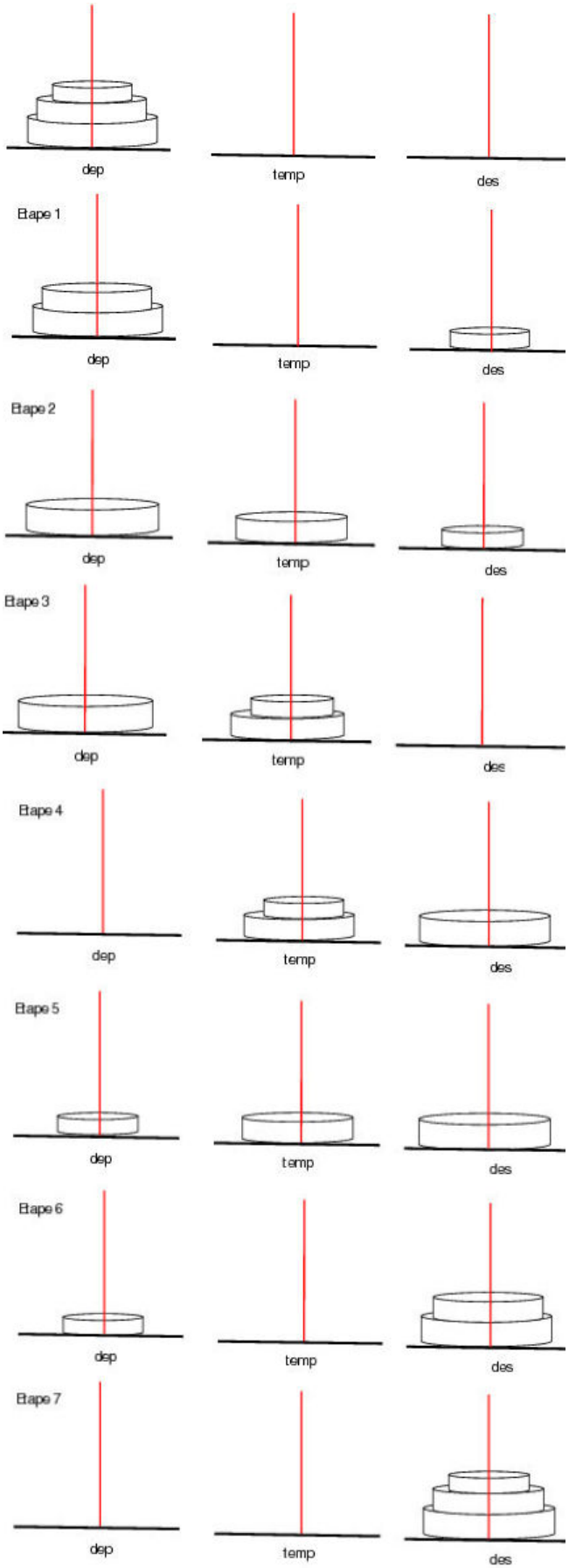


Figure 1.14 – Déroulement des étapes pour n=3 disques.

La Figure 1.15 montre une exécution de la procédure hanoi pour n=3.

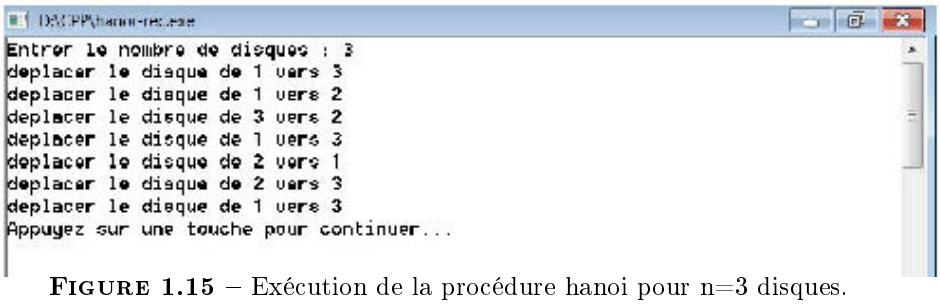


FIGURE 1.15 – Exécution de la procédure hanoi pour n=3 disques.

```
void hanoi(int n,int dep, int des, int temp)
{
if(n==1)
      printf("deplacer le disque %d de %d vers %d\n",n,dep,des) ;
      else
      {
       hanoi(n-1,dep,temp,des);
       hanoi(1,dep,des,temp);
       hanoi(n-1,temp,des,dep);
}
```

❀ Récursivité croisée ou mutuelle :

Deux sous-programmes sont mutuellement récursifs si l'un appelle l'autre et vice et versa.

☞ Exemple : un exemple typique est la parité d'un nombre entier n.

$$pair(n) = \begin{cases} vrai & si \;\; n = 0 \\ impair(n-1) & sinon \end{cases} \tag{1.5}$$

$$impair(n) = \begin{cases} faux & si \;\; n = 0 \\ pair(n-1) & sinon \end{cases} \tag{1.6}$$

Observer le code suivant de l'exemple qui produit une erreur en compilation puisque le problème qui se pose est laquelle des deux fonctions doit-on placer en premier **pair()** ou **impair()** puisque dans chacune d'elle il y'a un appel à l'autre ?

```
#include <stdio.h>
int pair(int n){
if(n==0)
    return 1; // La valeur 1 représente vrai et 0 représente faux
    else
    return impair(n-1);
}
int impair(int n){
if(n==0)
    return 0;
    else
    return pair(n-1);
}
int main(){
int n;
printf("Entrer la valeur de n :");
```

```
scanf("%d",&n);
if(pair(n)>0)
    printf("\nLe nombre %d est pair\n", n);
    else
    printf("\nLe nombre %d est impair\n", n);
}
```

D'où l'intérêt de l'utilisation des prototypes. La solution est de placer les prototypes et mettre les deux fonctions après la fonction principale.

```
#include <stdio.h>
int pair(int n);
int impair(int n);
int main(){
int n;
printf("Entrer la valeur de n :");
scanf("%d",&n);
if(pair(n)>0)
    printf("\nLe nombre %d est pair\n", n);
    else
    printf("\nLe nombre %d est impair\n", n);
}
int pair(int n){
if(n==0)
    return 1;
    else
    return impair(n-1);
}
int impair(int n){
if(n==0)
    return 0;
    else
    return pair(n-1);
}
```

Exemple de déroulement du code :

```
Entrer la valeur de n :7

pair(7)
Appel de impair(6)
impair(6)
Appel de pair(5)
pair(5)
Appel de impair(4)
impair(4)
Appel de pair(3)
pair(3)
Appel de impair(2)
impair(2)
Appel de pair(1)
pair(1)
Appel de impair(0)
impair(0)
Le nombre 7 est impair

--------------------------------
Process exited after 3.01 seconds with return value 0
Appuyez sur une touche pour continuer...
```

Figure 1.16 – Exemple de déroulement dans le cas d'une récursivité mutuelle.

1.6 Résumé

Ce chapitre a introduit la notion de sous-programme et la notion de récursivité. La récursivité simplifie considérablement certains codes puisqu'elle permet une écriture plus claire et plus raffinée des algorithmes. Ces derniers deviennent plus courts et plus faciles à maintenir mais toutefois, il est nécessaire d'analyser l'efficacité de l'algorithme récursif avant de l'adopter puisque l'appel de fonction/procédure a un coût non négligeable par rapport à un simple calcul arithmétique ou une opération d'affectation. Ainsi, la version récursive d'un algorithme peut être plus coûteuse que la version itérative, comme il a été observé avec le calcul des nombres de Fibonacci. Nous verrons par la suite que la notion de récursivité permet des approches du type « diviser pour règner » où le problème à résoudre est divisé en plusieurs sous-problèmes de même nature (de taille plus petite) de telle sorte que la résolution de ces sous-problèmes permettrait de résoudre le problème initial en entier en recombinant les solutions partielles.

Exercice 1 *: Sous-programmes et récursivité*

a/ Ecrire une fonction secondaire vérifiant si un nombre entier Nb est pair.

b/ Ecrire une fonction ***récursive*** *vérifiant si un nombre entier Nb est premier (un nombre premier est divisible uniquement par* 1 *et lui même).*

c/ En utilisant les deux fonctions précédentes, donner le code de la fonction principale (main) permettant de lire un tableau T de 100 valeurs entières et d'afficher le nombre d'éléments pairs et le nombre d'éléments premiers du tableau.

```
#include<stdio.h>
#define TAILLE 100
int premier(int Nb,int i) // La valeur initiale de i est 2
{if (i<Nb)  // Si i n'a pas encore atteint la valeur Nb
   if (Nb%i==0) // Si Nb est divisible par i alors il n'est pas premier
    return 0;
    else
    return premier(Nb,i+1); // Sinon Nb est redivisé encore par i+1
else
return 1;/*Dans ce cas i a atteint la valeur de Nb donc il n'est divisible que par 1 et lui même*/
}
int pair(int Nb)
{if(Nb%2==0)
   return 1;
   else
   return 0;
   }
int main(){
int T[TAILLE], i, n,nb_pairs=0, nb_premiers=0;
printf("Entrer le nombre d'elements du tableau : ");
scanf("%d",&n);
printf("Entrer les elements du tableau :\n");
for(i=0;i<n;i++)
scanf("%d",&T[i]);
// Calcul du nombre d'éléments premiers et du nombre d'éléments pairs
for(i=0;i<n;i++)
{
   if(premier(T[i],2) != 0) // Test pour les éléments premiers
              nb_premiers++;
```

```
   if(pair(T[i]) != 0) // Test pour les éléments pairs
          nb_pairs++;
}
printf("Nombre d'elements premiers = %d\nNbre d'elements pairs = %d\n", nb_premiers,nb_pairs);
}
```

La Figure 1.17 donne un exemple d'exécution du code.

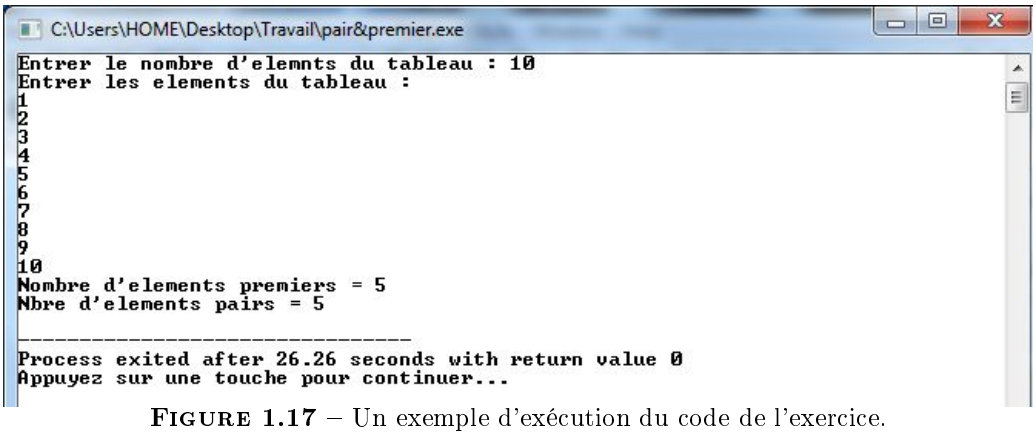


FIGURE 1.17 – Un exemple d'exécution du code de l'exercice.

Exercice 2 *calcul du nombre de combinaisons en utilisant la relation de Pascal :*

$$C_n^p = \begin{cases} 1 & si \;\; p=0 \;\; ou \;\; p=n \;\; \rightsquigarrow \;\; cas \;\; de \;\; base \\ C_{n-1}^p + C_{n-1}^{p-1} & sinon \;\; \rightsquigarrow \;\; cas \;\; général \end{cases} \tag{1.7}$$

```
#include<stdio.h>
unsigned long Comb(int n,int p)
{
   if(p==0 || p==n)
   return 1;
   else
   return Comb(n-1,p)+Comb(n-1,p-1);
}
int main(){
int n,p;
printf("Entrer n et p (avec p<=n) :");
scanf("%d%d",&n,&p);
printf("\nNombre de combinaisons de %d parmi %d = Comb(%d,%d) = %u\n",p,n,n,p,Comb(n,p));
}
}
```

Résultat de l'exécution :

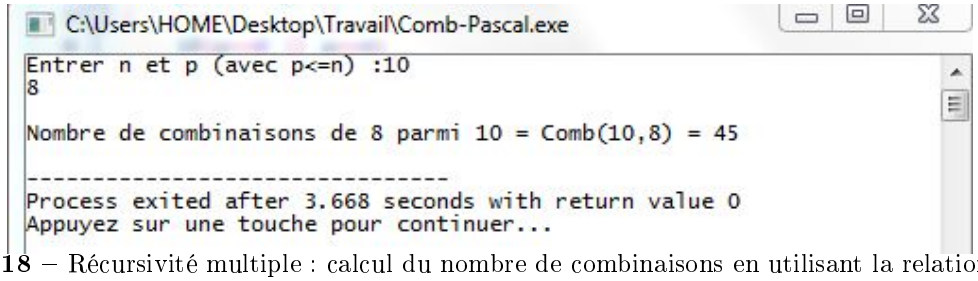


FIGURE 1.18 – Récursivité multiple : calcul du nombre de combinaisons en utilisant la relation de Pascal.

Le nombre de combinaisons peut être calculé aussi par la formule suivante :

$$C_n^p = \frac{n!}{p!(n-p)!} \tag{1.8}$$

En utilisant la fonction récursive fact(int n) vue précédemment, le code ci-dessous donne le même résultat.

```
#include<stdio.h>
unsigned long fact(int n)
{
   if(n==0)
   return 1;
   else
   return n*fact(n-1);
}
int main()
{int n,p;
unsigned long C;
printf("Entrer n et p (avec p<=n) :");
scanf("%d%d",&n,&p);
C=fact(n)/(fact(p)*fact(n-p));
printf("\nNombre de combinaisons de %d parmi %d = Comb(%d,%d) = %u\n",p,n,n,p,C);
}
```

Résultat de l'exécution :

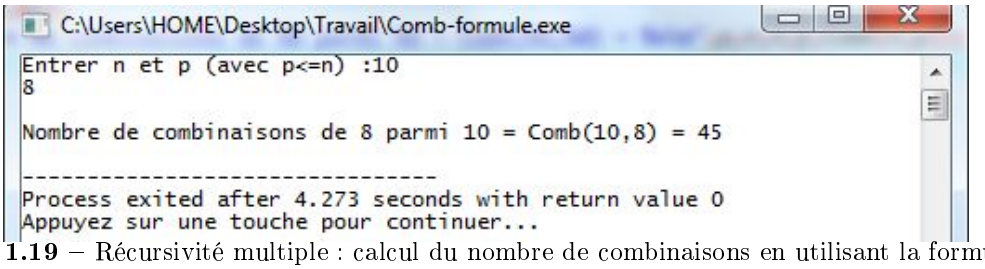


Figure 1.19 – Récursivité multiple : calcul du nombre de combinaisons en utilisant la formule de la factorielle.

La question qui se pose est lequel des deux codes est plus efficace ?

CHAPITRE 2

LES FICHIERS

Sommaire

Figures

2.1 Introduction

Jusqu'ici les programmes présentés dans ce manuel ont lus les données à partir du clavier (entrée standard) et affiché les résultats sur l'écran (sortie standard). Il arrive souvent que la quantité de données à lire soit importante et donc interagir avec le programme à chaque exécution pour introduire ces données au clavier devient lourd alors qu'il est possible de les récupérer directement à partir d'un support de stockage à chaque fois qu'il est nécessaire (de manière autonome). D'autre part, le caractère volatile de la mémoire centrale (RAM) nécessite le recours à une mémoire permanente pour sauver les résultats de crainte que ceux ci ne soient perdus à la mise hors tension du système. En plus, l'avantage de la capacité importante de stockage dont dispose les mémoires permanentes et leur coût favorable font d'elles le support le plus approprié pour le stockage à long terme. D'où l'intérêt **des fichiers** en lecture/écriture, où l'information peut y être stockée de façon permanente afin d'être réutilisée. Le mot **fichier** veut dire ensemble de fiches d'information qui est un objet physique stocké en général sur un support externe (secondaire), c'est à dire en dehors de la mémoire vive (RAM) comme une mémoire de masse (disque dur, disque optique, clé USB,...etc) assurant ainsi des entrées-sorties entre le programme et l'extérieur. Durant l'opération de lecture/écriture, l'information transite par une mémoire tampon (buffer en anglais) comprenant un certain nombre d'octets se trouvant dans la mémoire vive pour finir à la fin dans un fichier (cas d'une opération d'écriture) ou pour être lue par le programme (cas d'une opération de lecture) comme illustré dans Figure 2.1.

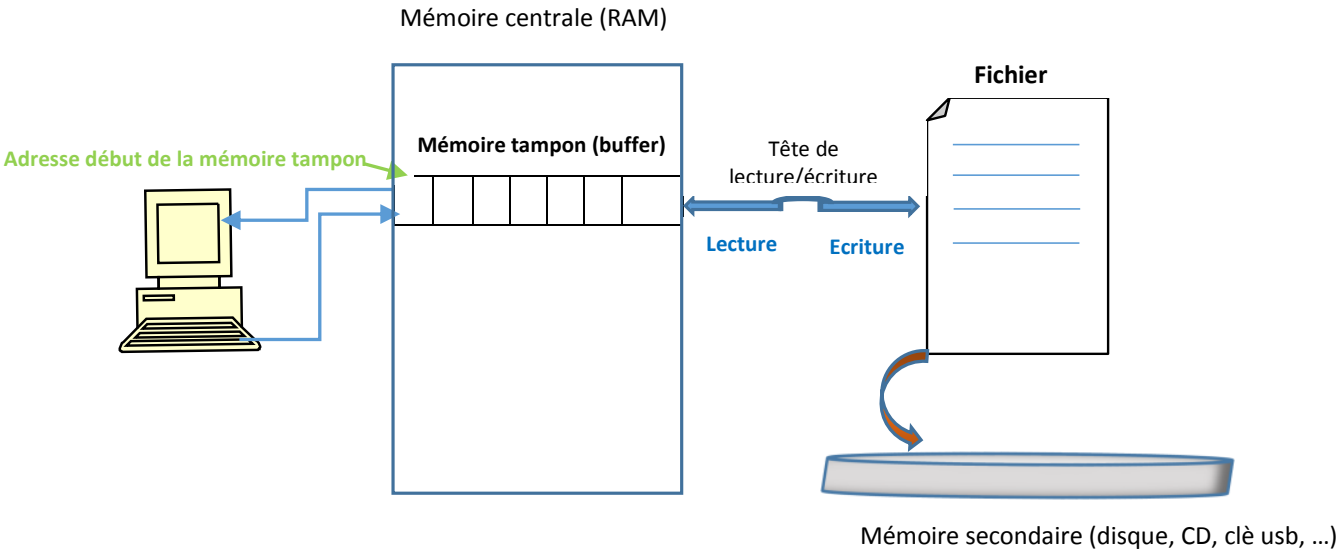


Figure 2.1 – Utilisation d'un fichier en lecture/écriture. Pour réguler les entrées-sorties entre le système et les mémoires de masse, l'information est mémorisée dans une zone tampon en attente de son envoi effectif vers le fichier se trouvant dans la mémoire de masse (cas d'écriture) ou vers le programme (cas de lecture).

Comme l'accès à une mémoire de masse est relativement plus long que l'accès à la mémoire vive, l'utilisation d'une zone tampon permet de réduire le nombre d'accès en lecture/écriture dans le fichier. Ainsi lors d'une opération d'écriture, le contenu de la mémoire tampon n'est vidé dans le fichier qu'une fois que celle ci est pleine. Il est toutefois possible que cette mémoire soit vidée (de manière forcée) dans certaines situations. On ne lit/écrit donc pas directement dans le fichier mais dans la mémoire tampon qui est allouée automatiquement à l'ouverture du fichier.

☺ Un programme manipulant un fichier a donc besoin d'un certain nombre d'informations comme l'adresse de l'emplacement de la mémoire tampon associée au fichier (pointeur vers cette zone mémoire), la position de la tête de lecture/écriture (position courante) et le mode d'accès au fichier (lecture ou écriture ou bien les deux à la fois). Le langage C, où un fichier est en général une suite d'octets fournit ces informations à travers une structure « **FILE** » définie dans la librairie <**stdio.h**>.

2.2 Définitions

Un fichier (file en anglais) est un ensemble d'informations enregistré sur une mémoire de masse (disque dur, disque optique, clé USB, bande magnétique, ...etc) et repérée par un nom physique qui est une chaîne de caractères. Les informations stockées ne sont pas forcément de même type, on peut avoir des caractères, des entiers, des réels, des structures, ...etc. L'accès aux informations peut être direct ou séquentiel. Pour être manipulé, un fichier doit avoir aussi un nom logique qui est un identificateur de variable de type pointeur vers la zone tampon associée au fichier, lequel n'a d'importance qu'à l'intérieur du programme où il a été défini.
La manipulation d'un fichier en lecture/écriture consiste à :

- ouvrir le fichier qui entraîne automatiquement la création d'une mémoire tampon dans la RAM gérée par un pointeur assurant l'allocation et la libération de la mémoire.
- lire/écrire dans le fichier ouvert qui revient à lire/écrire dans la mémoire tampon  (Berthet and Labatut, 2014a).
- fermer le fichier à la fin des opérations.

☺ La première opération sur les fichiers consiste tout d'abord à créer un lien entre le nom logique et le nom physique du fichier, ce qui est réalisé en langage C au moment de l'ouverture du fichier.

☺ Il faut faire la distinction entre ces deux noms qui ne sont pas nécessairement identiques. La durée de vie du nom logique expire à la fin du programme où il a été défini alors que le nom physique persiste et peut être réutilisé dans un autre programme et manipulé à l'aide d'un nom logique différent.

2.3 Types de fichiers

Tous les périphériques y compris l'écran et le clavier sont considérés comme fichiers ou flux (stream en anglais) de données, on peut les catégoriser comme suit :

Fichiers standards : cette catégorie regroupe les trois fichiers spéciaux définis par défaut pour tous les programmes :
- stdin (standard input) : le fichier d'entrée standard (clavier).
- stdout (standard output) : le fichier de sortie standard (écran).
- stderr (standard error) : le fichier d'affichage des messages d'erreur (par défaut, l'écran).

Fichiers texte (flux de type texte) : dans ce type de fichiers les informations sont mémorisées sous forme de chaînes de caractères (codées en ASCII) autrement dit du texte où un caractère (lettre, chiffre, symbole spécial) = un octet, organisé en lignes séparées par « $\backslash n$ » y compris les valeurs numériques (types int, float, double, ...etc). Ce mode est fait pour stocker uniquement les caractères affichables sur l'écran dont certains ont un usage particulier comme marquer la fin d'une ligne ou la fin du fichier. Ainsi, un fichier texte peut être lu ou modifié à partir de n'importe quel éditeur de texte.

Fichiers binaire (flux de type binaire) : dans ce type de fichiers, l'information est stockée sous forme d'une séquence d'octets les uns derrière les autres. Celle ci étant une recopie directe de la mémoire et donc non affichable. Les opérations de lecture/écriture sur ce type de fichiers sont plutôt rapides. Ce mode est surtout adapté aux données hétérogènes (mélange de types) et les structures complexes. Ce type de fichier contrairement à un fichier texte n'est ni éditable ni imprimable.

2.4 Manipulation des fichiers

Il est plus intéressent de décrire les primitives relatives aux fichiers directement en langage C plutôt qu'en utilisant un langage algorithmique puisque chaque langage a ses propres directives pour manipuler les fichiers. Comme il a été précisé auparavant, pour manipuler un fichier on déclare une variable de type pointeur sur la mémoire tampon associée au fichier en indiquant le mot clé **FILE**, structure prédéfinie dans la librairie <**stdio.h**> du langage. Appelons **fichier** cette variable qui est considérée comme étant le nom logique du fichier que va manipuler le programme, la syntaxe de déclaration est comme suit :

FILE *fichier ;

En général, les opérations qu'on peut réaliser sur un fichier sont :

- Ouverture d'un fichier ;
- Fermeture d'un fichier ;
- Lecture, écriture et déplacement dans un fichier ;
- Renommer un fichier ;
- Supprimer un fichier.

Rappelons donc que toutes ses opérations nécessitent la librairie <**stdio.h**>.

2.4.1 Ouverture d'un fichier

L'ouverture d'un fichier se fait à l'aide de la fonction **fopen** dont le prototype est :

File ***fopen**(const char *nom_physique, const char *mode) ;

Où :
nom_physique : est une chaîne de caractères spécifiant le nom avec lequel est enregistré le fichier sur la mémoire de masse.
mode : est une chaîne de caractères spécifiant le mode d'ouverture du fichier.
La fonction **fopen** renvoie un pointeur de type **FILE** dont la valeur doit servir pour lire et écrire dans le fichier.
☞ Exemple :

```
#include <stdio.h>
int main(){
FILE *infile; // Nom logique du fichier utilisé en entrée (lecture)
/* Mise en relation du nom logique avec le nom physique du fichier  qui est ouvert en mode lecture
    indiqué par "r" */
infile = fopen("C:/USTO-MB/INFO/Text.txt","r");
.........................
}
```

La variable **infile** est un pointeur vers l'adresse en mémoire de la zone tampon associée au fichier dont le nom physique est **Text.txt** situé dans le répertoire **INFO**, lui même situé dans le répertoire **USTO-MB** du disque dur. Le chemin d'accès au fichier dans la mémoire de masse est nécessaire pour que le système puisse le repérer, comme il apparaît dans la Figure 2.2.

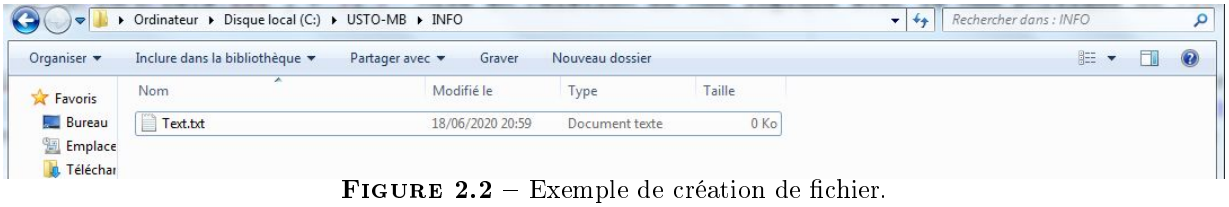


FIGURE 2.2 – Exemple de création de fichier.

☺ Il est toutefois préférable de définir le nom physique du fichier en utilisant la directive #**define** et utiliser dans le corps du programme le nom de la constante qui lui est associée au lieu du nom explicite du fichier.

☺ Dans le nom physique, l'extension du fichier précise son type (txt pour texte, exe pour exécutable, pdf pour portable document format,...etc.) et permet ainsi au système de sélectionner le programme qui peut ouvrir le fichier.

En reprenant l'exemple précédent, le code devient :

```
#include <stdio.h>
#define FILENAME "C:/USTO-MB/INFO/Text.txt"
int main(){
FILE *infile;
infile = fopen(FILENAME,"r");
.........................
}
```

Les différents modes d'ouverture d'un fichier sont :

"r" : le fichier est ouvert en lecture seule (read), il doit avoir été créé au préalable.

"w" : le fichier est ouvert en écriture seule (write). Si le fichier n'existe pas, il sera créé.

"a" : le fichier est ouvert en mode d'ajout (append). Ce mode permet de rajouter du texte en partant de la fin du fichier. Si le fichier n'existe pas, il sera créé.

"r+" : le fichier est ouvert en lecture et écriture. Il doit avoir été créé au préalable.

"w+" : le fichier est ouvert en lecture et écriture avec suppression du contenu au préalable. Si le fichier n'existe pas, il sera créé.

"a+" : le fichier est ouvert en mode ajout en lecture/écriture à la fin. Si le fichier n'existe pas, il sera créé.

☺ Le symbole « + » indique que le fichier est ouvert en mode mise à jour, recommandé pour des fichiers binaires dont l'ouverture nécessite l'ajout du caractère « b » après le premier caractère indiquant le mode d'ouverture ("rb","wb","ab","rb+","wb+","ab+"). Le mode d'usage de ces fichiers sera développé à la suite de ce chapitre.

Attention

☺ Au moment de l'utilisation de **fopen**, il faut vérifier si l'ouverture a été faite avec succès en testant la valeur du pointeur retourné. Si le pointeur vaut **NULL**, cela implique que l'ouverture du fichier a échoué et dans ce cas le traitement sur le fichier n'est pas possible (prévoir l'affichage d'un message d'erreur). Dans le cas contraire, c'est à dire le pointeur est différent de **NULL** (ouverture faite avec succès) alors le traitement peut être effectué.

☺ Il arrive toutefois que d'autres erreurs se produisent autres que l'erreur d'ouverture d'un fichier, comme par exemple lors d'une tentative d'opération de lecture/écriture (tentative d'écriture dans un fichier ouvert uniquement en lecture, tentative de lecture dans un fichier ouvert uniquement en écriture, tentative d'ajout ou de mise à jour dans un fichier fermé,...etc). Il y' a lieu de prévoir dans ces cas là un code spécifique à chaque éventuelle erreur.

☞ Exemple : ouverture d'un fichier en lecture et affichage d'un message en cas d'erreur.

```
#include <stdio.h>
#define FILENAME "C:/USTO-MB/INFO/Text.txt"
int main(){
FILE *infile;
infile = fopen(FILENAME,"r");
if(infile != NULL) // ou tout simplement if(infile)
  // Traitement sur le fichier
   else
// Message d'erreur en cas d'échec
printf("Le fichier %s n'a pas pu être crée!",FILENAME);
.........................
}
```

Ou bien :

```
#include <stdio.h>
#define FILENAME "C:/USTO-MB/INFO/Text.txt"
```

```
int main(){
FILE *infile;
if((infile = fopen(FILENAME,"r") != NULL)
  // Traitement sur le fichier
   else
// Message d'erreur en cas d'échec
printf("Le fichier %s n'a pas pu être crée!",FILENAME);
.........................
}
```

☺ En cas d'erreurs, la fonction **perror** permet d'afficher les messages d'erreurs sur la sortie d'erreurs standard, son prototype est :

void **perror**(const char *message) ;

Ainsi, le code précédent peut être écrit comme suit :

```
#include <stdio.h>
#define FILENAME "C:/USTO-MB/INFO/Text.txt"
int main(){
FILE *infile;
infile = fopen(FILENAME,"r");
if(infile != NULL)
  // Traitement sur le fichier
   else
// Message d'erreur en cas d'échec
perror("Le fichier n'a pas pu être crée!");
.........................
}
```

Le paramètre de la fonction **perror** doit être uniquement un message entre guillemets "", il ne peut donc y avoir d'arguments supplémentaires, comme ici l'argument spécifiant le nom du fichier qui a été omis.

2.4.2 Fermeture d'un fichier

Après son ouverture, un fichier doit être fermé après son utilisation pour libérer la mémoire occupée par la zone tampon qui lui est associée (elle sera donc supprimée de la mémoire vive). Cette opération annulant la liaison entre le nom physique du fichier et le nom logique est réalisée en utilisant la fonction **fclose** dont le prototype est :

int **fclose**(FILE *fichier) ;

Cette fonction prend comme paramètre le nom logique du fichier et renvoie une valeur de type **int** indiquant si la fermeture a été réalisée avec succès. Si la valeur retournée vaut :

- **0** : cela veut dire que le fichier a bien été fermé.
- **EOF** (pour l'anglais **E**nd **O**f **F**ile) : cela veut dire que la fermeture a échoué, cette valeur définie dans la librairie <**stdio.h**> indique une erreur ou que la fin du fichier est atteinte.

- La fermeture d'un fichier ouvert en écriture provoque le transfert de l'information restante dans la mémoire tampon dans le fichier, c'est pour cela que l'oubli de la fermeture du fichier entraîne la perte du reste de l'information qui était dans la mémoire tampon en attente d'être transférée dans le fichier.
- Il est possible de forcer l'écriture des données qui se trouvent dans la mémoire tampon (qui devient alors vide) en utilisant la fonction **fflush** qui retourne la valeur **0** en cas de succès et **EOF** en cas d'erreur, son prototype est :

  int **fflush**(FILE *fichier) ;
- Lors de la fermeture d'un fichier ouvert en écriture, la fin du fichier est marquée automatiquement par le symbole de fin de fichier **EOF**.
- Tester la fin de fichier revient à lire puis tester la valeur retournée par l'opération de lecture (si elle est égale à EOF ou non). La fonction **feof** qui est utilisée uniquement lors d'une opération de lecture ne sert pas à détecter la fin d'un fichier mais plus précisément si une lecture a échoué à cause de la rencontre de la fin de fichier, son prototype est :

  int **feof**(FILE *fichier) ;

  La fonction renvoie une valeur entière non nulle si la fin de fichier est atteinte.
- Tester si une erreur autre que « fin de fichier atteinte » s'est produite lors d'une opération de lecture/écriture sur le fichier revient à tester la valeur retournée par la fonction **ferror** dont le prototype est :

  int **ferror**(FILE *fichier) ;
- L'initialisation à zéro de l'indicateur d'erreur s'effectue par la fonction **clearerr** dont le prototype est :

  void **clearerr**(FILE *fichier) ;

2.4.3 Lecture et écriture dans un fichier texte

L'opération de lecture/écriture dans un fichier texte peut se faire de trois manières différentes selon l'information lue/écrite d'intérêt, si c'est :

- un caractère ;
- une chaîne de caractères ;
- une valeur numérique (types int, float, double, ...etc).

Si l'information est diverse (de type caractère et numérique), c'est le mode de lecture/écriture formatée qui est privilégié afin de distinguer les différents types.

2.4.3.1 Écriture dans un fichier texte

Pour écrire dans un fichier texte, il y' a trois possibilités :

❀ **fputc** : écrit un caractère dans le fichier (un seul caractère à la fois), son prototype est :

int **fputc**(int caractere, FILE *fichier) ;

Où caractere est une constante ou variable de type caractère et fichier est le nom logique du fichier. La fonction retourne **une valeur entière** (comprise entre 0 et 255) correspondant au caractère lu ou la constante **EOF** en cas d'erreur.

```
putchar(c) ; // équivalente à fputc(c,stdout);
```

☞ Exemple : écriture d'un caractère lu en entrée dans un fichier texte TEXT.txt créé au préalable. Le fichier tel qu'il apparaît dans Figure 2.3 est ouvert en lecture/écriture (mode mise à jour).

Figure 2.3 – Fichier TEXT.txt avant traitement.

```
#include <stdio.h>
#define FILENAME "TEXT.txt"
int main(){
FILE *fichier;
char c;
fichier = fopen(FILENAME, "r+");
if(fichier != NULL )
{
c=getchar(); //c correspond à un appui sur une touche du clavier
fputc(c,fichier); //Ecriture du caractère c dans le fichier
fclose(fichier);
}
else
printf("Impossible d'ouvrir le fichier %s\n",FILENAME);
}
```

Résultat de l'exécution :

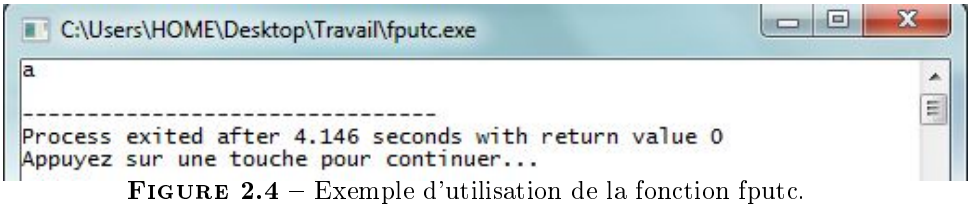


Figure 2.4 – Exemple d'utilisation de la fonction fputc.

Figure 2.5 – Fichier TEXT.txt après exécution du code.

Le premier caractère dans le fichier est remplacé par la lettre 'a' et donc "Bonjour" est devenue "aonjour". Il est possible d'éviter d'écraser le contenu du fichier en l'ouvrant en mode ajout en lecture/écriture "a+" et le résultat sera comme illustré dans la figure 2.6.

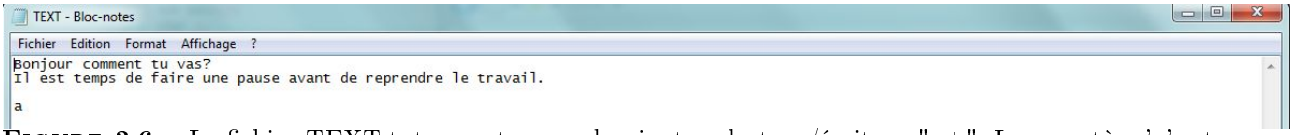
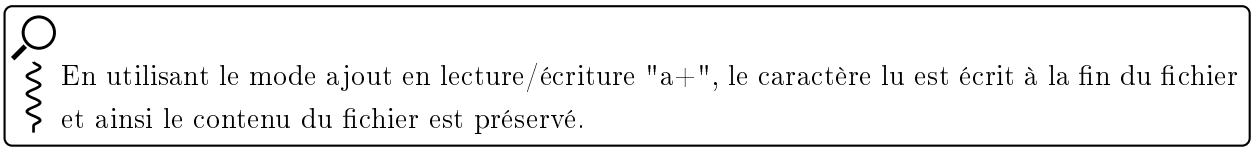


FIGURE 2.6 – Le fichier TEXT.txt ouvert en mode ajout en lecture/écriture "a+". Le caractère 'a' est ajouté en fin de fichier.


En utilisant le mode ajout en lecture/écriture "a+", le caractère lu est écrit à la fin du fichier et ainsi le contenu du fichier est préservé.

❀ **fputs** : écrit une chaîne de caractères dans le fichier terminée par un retour à la ligne, son prototype est :

char* **fputs**(const char *chaine, FILE *fichier) ;

Où chaine est la chaîne de caractères à écrire dans le fichier. La fonction renvoie EOF en cas d'erreur.

☞ Exemple : écriture d'une chaîne de caractère dans un fichier ouvert en écriture.

```
#include <stdio.h>
#define FILENAME "Essai.txt"
int main(){
FILE *fichier;
fichier = fopen (FILENAME, "w");
if(fichier != NULL )
{
//Ecriture de la chaîne dans le fichier
fputs("Salut tout le monde!", fichier);
fclose(fichier);
}
else
{
printf("Impossible d'ouvrir le fichier %s\n",FILENAME);
return 1;
}}
```

Résultat de l'exécution :

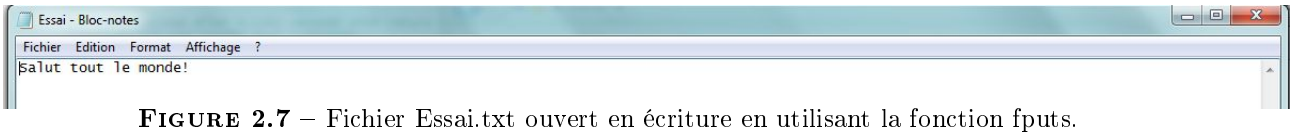


FIGURE 2.7 – Fichier Essai.txt ouvert en écriture en utilisant la fonction fputs.

❀ **fprintf** : écrit une chaîne formatée dans le fichier, le nom logique du fichier est spécifié comme premier paramètre dans le prototype de la fonction dont l'utilisation est similaire à **printf**.

fprintf(fichier,"format",expression$_1$, ...,expression$_n$)

Où les spécifications de format sont les mêmes que pour **printf**.

☞ Exemple1 : ouverture d'un fichier en écriture avec message d'erreur en cas de difficultés.

```c
#include <stdio.h>
int main(){
FILE *outfile;
outfile = fopen("Exemple1.txt", "w");
if(outfile != NULL) // Ouverture avec succès
{
 // Ecriture formatée sur le fichier
fprintf(outfile, "Bonjour tout le monde!\n"); /*Flux dirigé vers le fichier Exemple1.txt*/
fclose(outfile); // Fermeture du fichier
return 0; // Déroulement normal (avec succès)
}
 else {
fprintf(stderr,"Erreur d'ouverture du fichier !\n");/*Flux dirigé vers le fichier stderr (écran)*/
printf("Erreur d'ouverture du fichier !\n");/*Même effet que l'instruction précédente*/
perror("Erreur d'ouverture du fichier !\n");/*Même effet que les 2 instructions précédentes*/
return 1; // Déroulement anormal (échec de l'ouverture du fichier)
}}
```

Résultat de l'exécution :

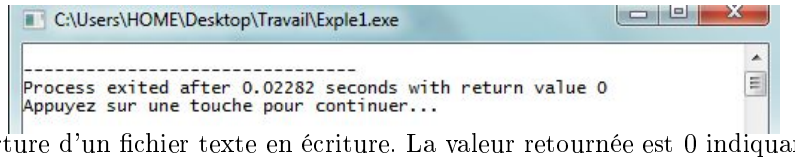


Figure 2.8 – Ouverture d'un fichier texte en écriture. La valeur retournée est 0 indiquant un déroulement normal du traitement.

Figure 2.9 – Fichier texte résultat ouvert en écriture.

☞ Exemple2 : ouverture d'un fichier en écriture pour y enregistrer les n premiers nombres entiers, n lu en entrée.

```c
#include <stdio.h>
int main(){
FILE *outfile;
int n,i;
printf("Entrer la valeur de n :");
scanf("%d",&n);
// Ouverture du fichier en écriture
outfile = fopen("Essai.txt", "w");
if(outfile != NULL){
// Ecriture formatée sur le fichier
for(i=1;i<=n;i++)
fprintf(outfile,"%d\t",i); // Flux dirigé vers le fichier Essai.txt
fclose(outfile); // Fermeture du fichier
return 0; // Déroulement normal
}
else
return 1;//Affichage de 1 pour indiquer l'échec de l'ouverture
}
```

Résultat de l'exécution :

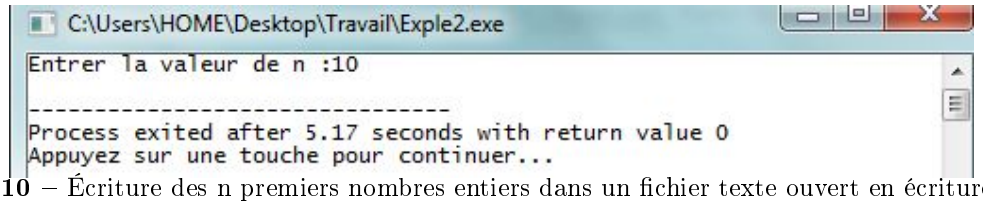


FIGURE 2.10 – Écriture des n premiers nombres entiers dans un fichier texte ouvert en écriture, n lu en entrée.

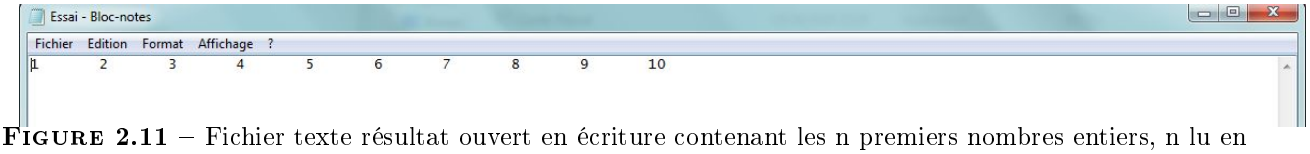


FIGURE 2.11 – Fichier texte résultat ouvert en écriture contenant les n premiers nombres entiers, n lu en entrée.

☞ Exemple3 : écriture formatée d'un tableau de structure Etudiant dans un fichier ouvert en écriture.

```
#include <stdio.h>
#define FILENAME "Data.txt"
typedef struct{
char nom[30], prenom[30];
int age;
}Etudiant;
int main(){
FILE *fichier;
Etudiant e[5];
int i;
fichier = fopen(FILENAME, "w");
if(fichier != NULL )
{
for(i=0;i<5;i++)
{
// Saisie des informations
printf("Entrer le nom : ");
scanf("%s",e[i].nom);
printf("Entrer le prenom : ");
scanf("%s",e[i].prenom);
printf ("Entrer l'age : ");
scanf("%d", &e[i].age );
//Ecriture des informations dans le fichier
fprintf(fichier, "%s\t%s\t%d\n",e[i].nom,e[i].prenom,e[i].age);
}
fclose(fichier);
}
else
printf ("Impossible d'ouvrir le fichier %s en écriture!\n",FILENAME);
}
```

Résultat de l'exécution :

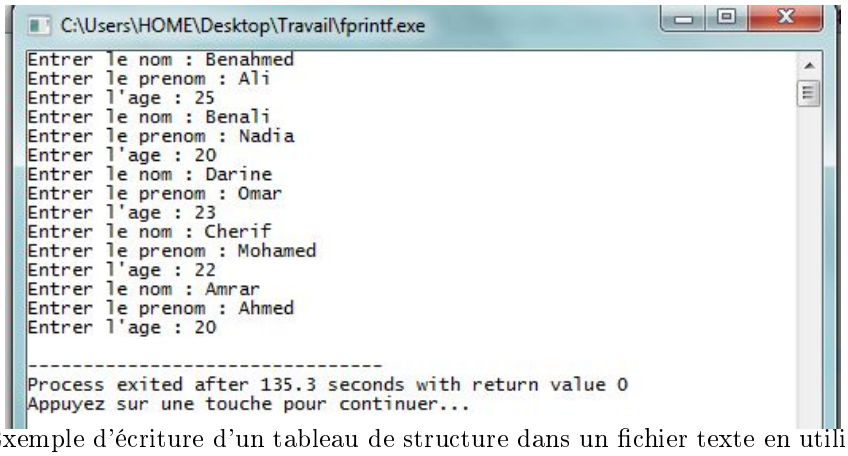


FIGURE 2.12 – Exemple d'écriture d'un tableau de structure dans un fichier texte en utilisant la fonction fprintf.

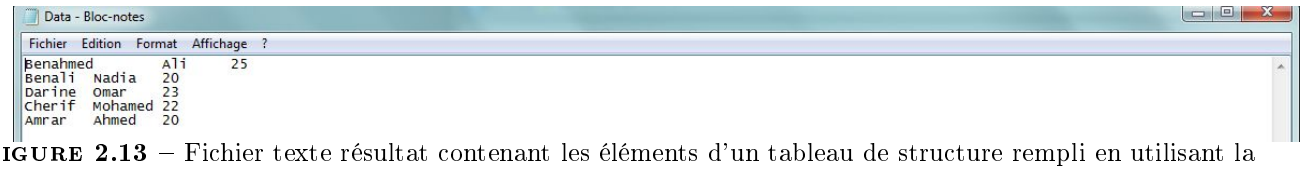


FIGURE 2.13 – Fichier texte résultat contenant les éléments d'un tableau de structure rempli en utilisant la fonction fprintf.

2.4.3.2 Lecture dans un fichier texte

De la même manière que pour l'écriture, les trois fonctions sont :

❀ **fgetc** : lit un caractère à la fois et se prépare à lire le caractère suivant, son prototype est :

int **fgetc**(FILE *fichier) ;

```
c=getchar() ; // équivalente à c=fgetc(stdin);
```

☞ Exemple : lecture caractère par caractère du fichier texte Exemple1.txt (voir Figure 2.9).

a/ Opération réalisée par la fonction principale main() :

```
#include <stdio.h>
#define FILENAME "Exemple1.txt"
int main(){
FILE *infile;
int c;
infile = fopen(FILENAME, "r");
if(infile!=NULL)
//Utilisation d'une boucle do while
{ do
   {c=fgetc(infile);
    printf("%c",c);
    } while (c!=EOF);
    fclose(infile);
}
else
fprintf(stderr,"Erreur d'ouverture du fichier %s en lecture!\n",FILENAME) ;
}
```

b/ Opération réalisée par une fonction secondaire :

Dans ce cas le nom logique du fichier est passé en paramètre à la fonction secondaire.

```
#include <stdio.h>
#define FILENAME "Exemple1.txt"
void lireTexte(FILE *infile){
int c;
// Utilisation d'une boucle while
while((c=fgetc(infile))!=EOF)
    printf("%c",c);
}
int main(){
FILE *infile = fopen(FILENAME,"r");
if(infile != NULL)
{
   lireTexte(infile);
   fclose(infile);
}
else
fprintf(stderr,"Erreur d'ouverture du fichier %s en lecture!\n",FILENAME) ;
}
```

Résultat de l'exécution :

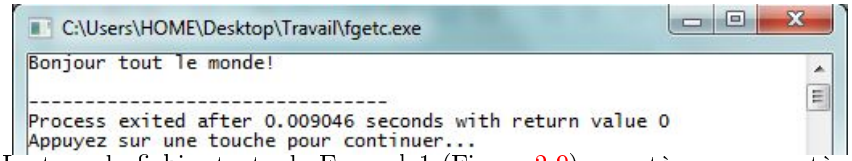


FIGURE 2.14 – Lecture du fichier texte de Exemple1 (Figure 2.9) caractère par caractère en utilisant la fonction fgetc.

❀ **fgets** : lit une chaîne et renvoie **NULL** en cas d'échec, son prototype est :

char ***fgets**(const char *chaine, int nombre, FILE *fichier) ;

Où nombre est le nombre maximum de caractères à lire. La fonction lit au maximum une ligne de texte, elle s'arrête au premier « $\backslash n$ » (retour à la ligne) qu'elle rencontre.

☞ Exemple : ouverture du fichier TEXT.txt créé auparavant (voir Figure 2.6) en lecture :

```
#include <stdio.h>
#define TMAX 500 // Longueur max de la chaîne de caractères
int main(){
FILE *infile;
char chaine[TMAX];
infile = fopen("TEXT.txt", "r");
if (infile != NULL )
{
while(fgets(chaine,TMAX,infile) != NULL )
     printf("%s",chaine);
fclose(infile);
}
else
perror("Oups erreur d'ouverture du fichier en lecture!");
}
```

Résultat de l'exécution :

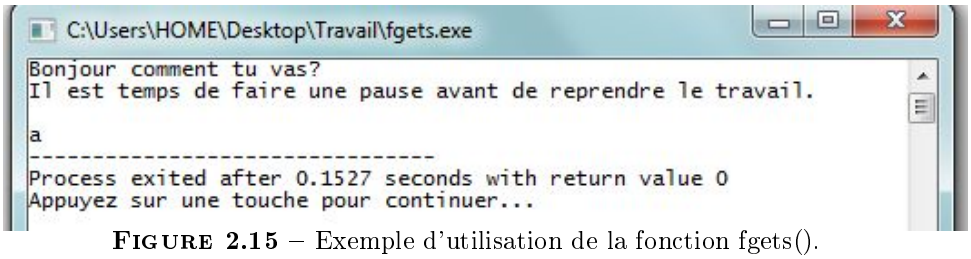


FIGURE 2.15 – Exemple d'utilisation de la fonction fgets().

❀ **fscanf** : lit une chaîne formatée, le nom logique du fichier est le premier argument de la fonction.

fscanf(fichier,"format",argument$_1$, ...,argument$_n$)

Les spécifications de format sont les mêmes que pour **scanf**.

☞ Exemple1 : ouverture du fichier Essai.txt crée auparavant en lecture pour lire les nombres qui y sont stockés et test de fin de fichier.

```
#include <stdio.h>
int main(){
FILE* infile;
int i;
infile = fopen("Essai.txt", "r");
if(infile != NULL)
 {
 while(!feof(infile)) // Test de fin de fichier
  {
   fscanf(infile,"%d",&i);//Flux lu à partir du fichier Essai.txt
   printf("%d\t",i); //Flux dirigé vers l'écran
  }
 fclose(infile);
 }
 else
 {
  printf("Impossible d'ouvrir le fichier en lecture\n");
   return 1 ;
}}
```

Résultat de l'exécution :

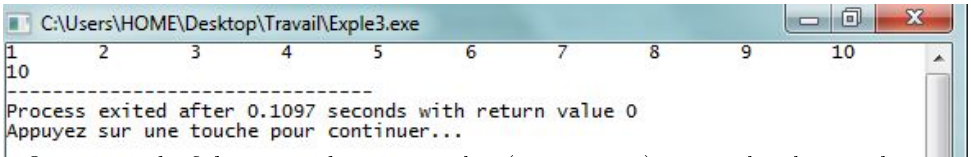


FIGURE 2.16 – Ouverture du fichier créé dans Exemple2 (Figure 2.11) pour y lire les nombres entiers stockés.

Remarquer que la dernière valeur 10 est affichée deux fois, ce qui montre qu'avec la fonction **feof** même si la fin de fichier est atteinte, le bloc d'opérations dans le while s'exécute comme même encore une dernière fois. En fait, la fonction **feof** permet juste de savoir si la dernière opération sur le fichier a échoué à cause d'une fin de fichier.

Reprenons ce code avec une légère modification :

```
#include <stdio.h>
int main(){
FILE* infile;
int i,c; //c valeur retournée par la fonction fscanf
infile = fopen("Essai.txt", "r");
if(infile!=NULL)
  {
  while((c=fscanf(infile,"%d",&i)) != EOF) //Test de fin de fichier
    printf("%d\t",i); // Flux dirigé vers l'écran
  fclose(infile);
  }
  else
   {
  printf("Impossible d'ouvrir le fichier en lecture\n");
  return 1;
   }}
```

Résultat de l'exécution :

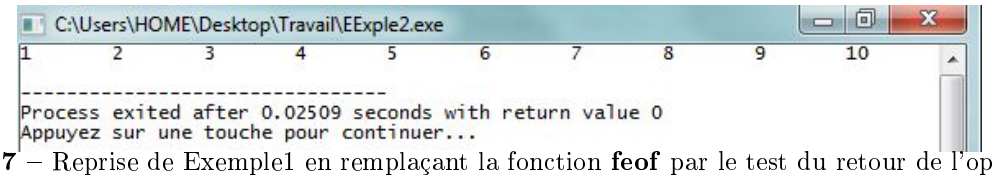


FIGURE 2.17 – Reprise de Exemple1 en remplaçant la fonction **feof** par le test du retour de l'opération de lecture par rapport à la constante EOF.

Remarquer que la dernière valeur 10 est affichée une seule fois. Ainsi, en testant directement la valeur lue sur le fichier par rapport à la constante EOF, une fois la fin de fichier atteinte le bloc d'opérations dans le while ne s'exécute plus. Donc, pour contrôler la fin de fichier lors d'une opération de lecture mieux vaut tester le retour de la fonction de saisie directement. La valeur NULL indique un cas d'erreur et la valeur EOF indique la fin de fichier.

Reprenons maintenant Exemple2 (Figure 2.11) et Exemple1 dans un même programme et utilisons la directive #**define** pour pouvoir afficher le nom physique du fichier en cas d'erreurs, ainsi le code devient :

```
#include <stdio.h>
#define FILENAME "Essai.txt"
int main(){
FILE *infile,*outfile;
int n,i,c;
printf("Entrer la valeur de n :");
scanf("%d",&n);
outfile = fopen(FILENAME, "w"); //Ouverture du fichier en écriture
if(outfile!=NULL){
 // Ecriture formatée sur le fichier
for(i=1;i<=n;i++)
  fprintf(outfile,"%d\t",i); //Flux dirigé vers le fichier Essai.txt
fclose(outfile); //Fermeture du fichier
```

```
}
else
{
printf("Impossible d'ouvrir le fichier %s en écriture\n",FILENAME);
return 1; //Echec d'ouverture en écriture de Essai.txt
}
infile = fopen(FILENAME, "r"); //Ouverture du même fichier en lecture
if(infile!=NULL){
 while( (c=fscanf(infile,"%d",&i)) != EOF) // Test de fin de fichier
  printf("%d\t",i); //Flux dirigé vers l'écran
 fclose(infile);
 printf("\nParfait tout s'est bien passe!\n");
 return 0; // Tout s'est bien passé
}
else
{
printf("Impossible d'ouvrir le fichier %s en lecture\n",FILENAME);
return 2; //Echec d'ouverture en lecture de Essai.txt
}}
```

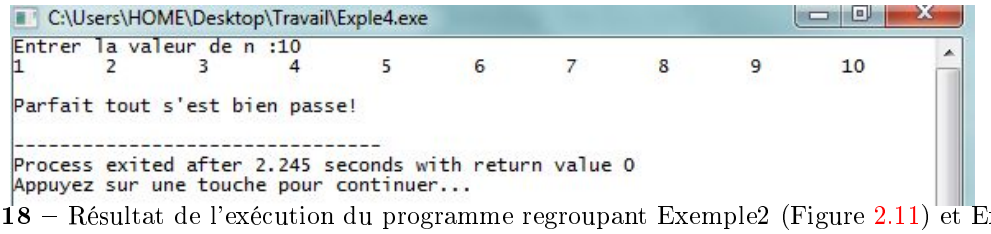


FIGURE 2.18 – Résultat de l'exécution du programme regroupant Exemple2 (Figure 2.11) et Exemple1.

☞ Exemple2 : lecture et affichage des informations stockées dans le fichier Data.txt créé auparavant dont le contenu apparaît dans Figure 2.13.

```
#include <stdio.h>
#define FILENAME "Data.txt"
typedef struct{
char nom[30],prenom[30];
int age;
}Etudiant;
int main(){
FILE *fichier ;
Etudiant e[5];
int i;
fichier = fopen(FILENAME, "r");
if(fichier != NULL){
for(i=0;i<5;i++)
{
// Lecture des informations dans le fichier
fscanf(fichier, "%s\t%s\t%d\n",&e[i].nom ,&e[i].prenom ,&e[i].age);
printf("%s\t%s\t%d\n",e[i].nom ,e[i].prenom ,e[i].age);
}
fclose(fichier);}
else
printf("Impossible d'ouvrir le fichier %s!\n", FILENAME );
}
```

Résultat de l'exécution :

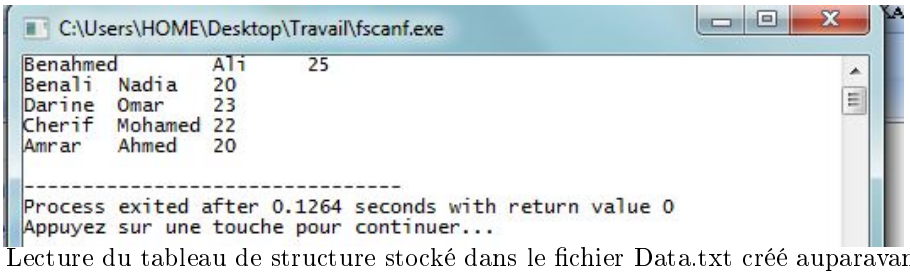


FIGURE 2.19 – Lecture du tableau de structure stocké dans le fichier Data.txt créé auparavant en utilisant fscanf.

2.4.4 Lecture et écriture dans un fichier binaire

Comme dans un fichier binaire l'information est recopiée telle qu'elle a été codée en mémoire centrale, la lecture/écriture de celle ci se fait uniquement par programme à l'aide de fonctions spécifiques qui permettent l'accès par bloc d'information. Ces fonctions ont comme paramètres l'adresse de la zone tampon où récupérer/stocker les données à lire/écrire, la taille en octets d'un élément de données et le nombre d'éléments à lire/écrire. La zone tampon peut être un tableau comme elle peut être une variable simple ou structurée. L'accès par bloc d'information dans un fichier binaire est illustré dans Figure 2.20.

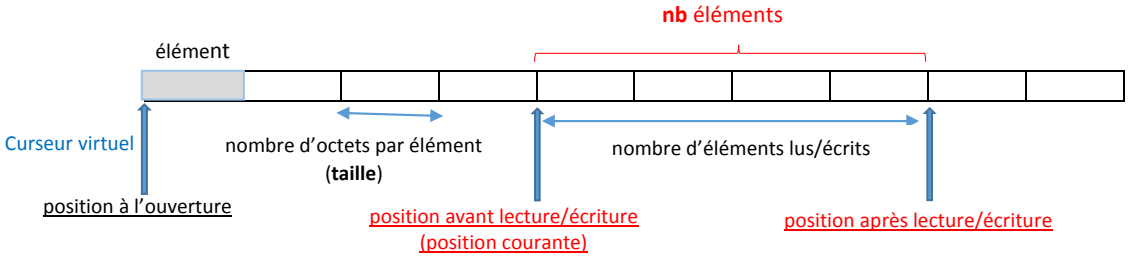


FIGURE 2.20 – Accès par bloc dans un fichier binaire. Le curseur virtuel indique la position dans le fichier.

2.4.4.1 Lecture d'un bloc de données dans un fichier binaire

La lecture d'un bloc d'information se fait en utilisant la fonction **fread** dont le prototype est :

int **fread**(const *void tampon, int taille, int nb, FILE *fichier) ;

La fonction lit un nombre **nb** d'éléments de taille **taille** octets à partir de la position courante du fichier indiqué par son nom logique **fichier** vers l'adresse début de la zone mémoire désignée par **tampon** (adresse du premier octet de la liste d'éléments à lire se trouvant dans la mémoire vive) et retourne le nombre d'éléments effectivement lus qui est égal à **nb** en cas de succès et inférieur à **nb** en cas d'erreur ou fin de fichier.

☞ Exemple1 : lecture d'une variable simple.

```
.........................
fread(&a,sizeof(int),1,fichier); // &a adresse d'une variable simple entière
fread(&c,sizeof(char),1,fichier); // &c adresse d'une variable simple de type caractère
```

☞ Exemple2 : lecture d'un tableau de valeurs. Comme le nom d'un tableau est un pointeur vers sa première case (son premier élément), le symbole « & » est omis.

```
#include <stdio.h>
#include <string.h>
int main()
{
 FILE *fichier = fopen("Essai.dat", "rb");
 if(fichier != NULL)
 {
 int n, i, T[20];
 n = fread(T, sizeof(T[0]),sizeof(T)/sizeof(T[0]),fichier);/* ce qui est encadré est le nombre d'élé
    ments de T et n est le nombre d'éléments de T effectivement lus */
// Affichage des éléments de T lus dans le fichier
 for(i=0; i<n;i++)
 printf("%d\n",T[i]);
 fclose(fichier);
 }
else
{
perror("Oups erreur!");
return 1;
}}
```

2.4.4.2 Écriture d'un bloc de données dans un fichier binaire

L'écriture d'un bloc d'information se fait en utilisant la fonction **fwrite** dont le prototype est :

int **fwrite**(const *void tampon, int taille, int nb, FILE *fichier) ;

La fonction écrit un nombre **nb** d'éléments de taille **taille** octets à partir de l'adresse début de la zone tampon indiquée par **tampon** (adresse du premier octet de la liste d'éléments à écrire se trouvant dans la mémoire vive) vers le fichier et retourne le nombre d'éléments effectivement écrits.
☞ Exemple1 : écriture d'une variable simple dans un fichier.

```
.........................
fwrite(&a,sizeof(int),1,f); // &a adresse d'une variable simple entière
fwrite(&c,sizeof(char),1,f); // &c adresse d'une variable simple de type caractère
```

☞ Exemple2 : écriture d'une chaîne de caractères dans un fichier binaire.

```
#include <stdio.h>
#include <string.h>
int main()
{
 FILE *fichier;
 char chaine[] = "Bonjour tout le monde!";
```

```
 f = fopen("Text.txt", "wb"); // Fichier binaire ouvert en écriture
 if (fichier != NULL)
 {
 fwrite(chaine, sizeof(char), strlen(chaine), fichier);
 fclose(fichier);
 }
 else
{
 perror("Oups erreur!");
 return 1;
}}
```

☞ Exemple3 : écriture d'un tableau dans un fichier.

```
#include <stdio.h>
int main()
{char T[] = {'B','O','N','J','O','U','R'}; // sizeof(T) taille totale allouée pour T
int V[4] ={10,20,30,40};// sizeof(V) taille totale allouée pour V
float R[3] = {12.5,3.5,5};// sizeof(R) taille totale allouée pour R
 FILE * fichier = fopen("Sortie.dat", "wb"); // Fichier binaire ouvert en écriture
 if(fichier != NULL)
 {
//Affichage du nombre d'octets par élément pour chaque tableau
printf("\nsizeof(T[0])=%d octet", sizeof(T[0]));
printf("\nsizeof(V[0])=%d octets", sizeof(V[0]));
printf("\nsizeof(R[0])=%d octets", sizeof(R[0]));
//Ecriture des trois tableaux dans le fichier
fwrite(T, sizeof(T[0]), sizeof(T)/sizeof(T[0]), fichier);
fwrite(V, sizeof(V[0]), sizeof(V)/sizeof(V[0]), fichier);
fwrite(R, sizeof(R[0]), sizeof(R)/sizeof(R[0]), fichier);
fclose(fichier);
 }
else
{
perror("Oups erreur d'ouverture du fichier en écriture!");
return 1 ;
}}
```

Résultat de l'exécution :

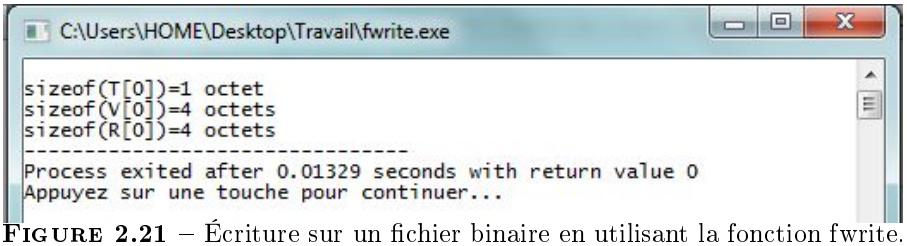


Figure 2.21 – Écriture sur un fichier binaire en utilisant la fonction fwrite.

Visualisation du fichier créé :

Figure 2.22 – Fichier binaire ouvert en écriture. Bien entendu, les octets d'information qui y sont stockés ne sont pas lisibles.

Reprenons le même fichier pour afficher par programme son contenu :

```
#include <stdio.h>
int main()
{char T[7];
int V[4],i,n;
float R[3];
 FILE * fichier = fopen("Sortie.dat", "rb"); /*Fichier binaire ouvert en écriture*/
 if (fichier != NULL)
 {
n=fread(T, sizeof(char),7, fichier);
printf("\nEléments du premier tableau :");
for(i=0;i<n;i++)
printf("%c",T[i]);
n=fread(V, sizeof(int),4, fichier);
printf("\nEléments du deuxième tableau :");
for(i=0;i<n;i++)
printf("%d\t",V[i]);
n=fread(R, sizeof(float),3, fichier);
printf("\nEléments du troisième tableau :");
for(i=0;i<n;i++)
printf("%.1f\t",R[i]);
fclose(fichier);
 }
else
{
perror("Oups erreur!");
return 1 ;
}}
```

Résultat de l'exécution :

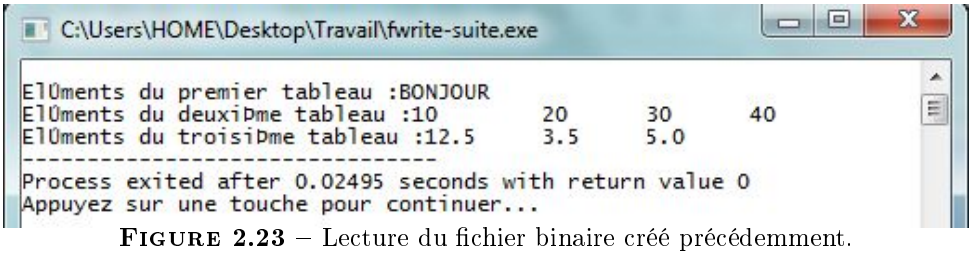


Figure 2.23 – Lecture du fichier binaire créé précédemment.

☞ Exemple4 : saisie des information d'une structure Etudiant.

a/ Utilisation d'un fichier binaire.

```
#include <stdio.h>
#define nb 3
typedef struct{
   char nom[20],prenom[40];
   int code,age;}Etudiant;
int main(){
Etudiant e[nb],v[nb];
FILE *outfile;
int i;
//Affichage du nombre d'octets alloué à la structure Etudiant
```

```
printf("\nsizeof(Etudiant) = %d octets\n",sizeof(Etudiant));
outfile = fopen("etudiant","wb");
if(outfile != NULL)
{for(i=0;i<nb;i++)
 {
    printf("\nEntrer le nom : ");
    scanf("%s",e[i].nom);
    printf("\nEntrer le prenom : ");
    scanf("%s",e[i].prenom);
    printf("\nEntrer le code : ");
    scanf("%d",&e[i].code);
    printf("\nEntrer l'age : ");
    scanf("%d",&e[i].age);
   }
fwrite(e,sizeof(Etudiant),nb,outfile);
fclose(outfile);
}
else
return 1; //En cas d'échec d'ouverture du fichier en écriture
outfile = fopen("etudiant","rb");
if(outfile != NULL)
{
fread(v,sizeof(Etudiant),nb,outfile);
for(i=0;i<nb;i++)
 printf("\n%s  %s  %d  %d",v[i].nom,v[i].prenom,v[i].code,v[i].age);
fclose(outfile);
}
else
return 2;//En cas d'échec d'ouverture du fichier en lecture
}
```

Résultat de l'exécution :

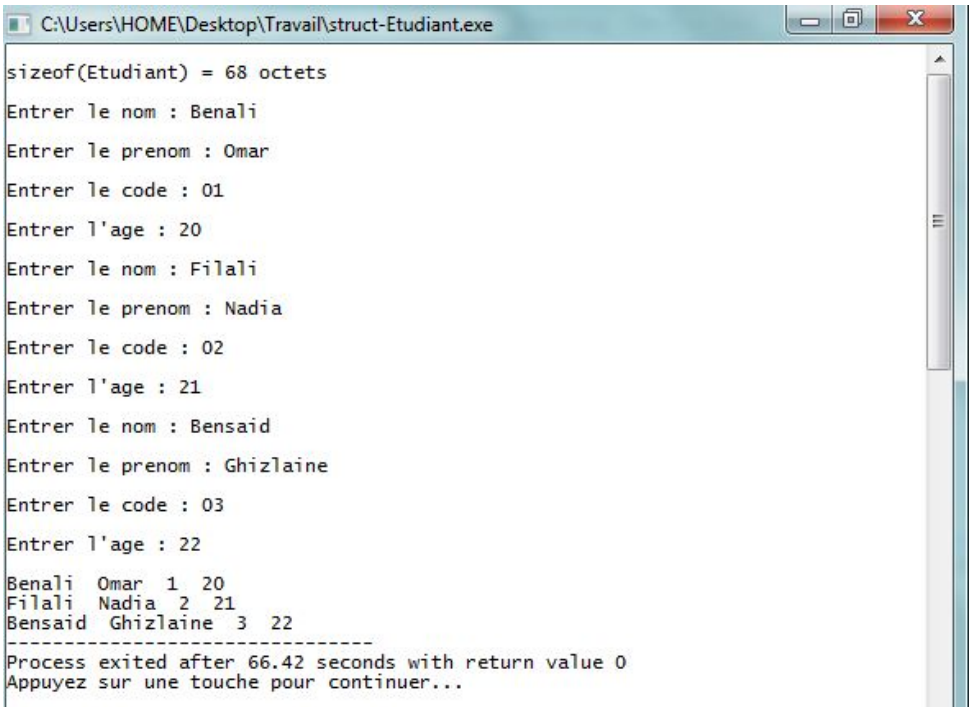


Figure 2.24 – Ecriture et lecture d'une structure dans et à partir d'un fichier binaire. Le nombre d'octets alloués pour la structure est 68 soit un total de 60 octets pour le nom et prenom, 4 octets pour le code (int) et 4 octets pour l'age (int).

b/ Utilisation d'un fichier texte.

```
#include <stdio.h>
#include <string.h>
#define nb 5
typedef struct {
   char nom[20],prenom[40];
   int code,age;
}Etudiant;
int main()
{
   Etudiant e[nb];
   FILE *outfile;
   int i;
   outfile = fopen("etudiant","w");
   if(outfile != NULL)
   {
   for(i=0;i<nb;i++)
     {
      printf("\nEntrer le nom : ");
      scanf("%s",e[i].nom);
      fprintf(outfile,"%s",e[i].nom);
      printf("\nEntrer le prenom : ");
      scanf("%s",e[i].prenom);
      fprintf(outfile,"%s",e[i].prenom);
      printf("\nEntrer le code : ");
      scanf("%d",&e[i].code);
      fprintf(outfile,"%d",e[i].code);
       printf("\nEntrer l'age : ");
      scanf("%d",&e[i].age);
      fprintf(outfile,"%d",e[i].age);
     }
   /*Comme il est possible d'écrire sur le fichier une seule fois à ce niveau comme suit :
   fprintf(outfile,"%s\t%s\t%d\t%d",e[i].nom,e[i].prenom,e[i].code,e[i].age);*/
     fclose(outfile);
}
else
return 1;
}
```

☞ Exemple5 : Ouverture du fichier binaire de la structure Etudiant créé dans l'exemple précédent en mode mise à jour. Supposons que l'on veuille changer le prénom du deuxième étudiant dans le fichier, comment procéder ?

Le code suivant modifie le dernier élément et le re-écrit à la fin du fichier.

```
#include <stdio.h>
#include <string.h>
typedef struct {
   char nom[20],prenom[40];
   int code,age;
}Etudiant;
int main()
{
   Etudiant e;
```

```
   FILE *outfile;
   int i;
   outfile = fopen("etudiant","rb+");
if(outfile != NULL)
{ while(fread(&e,sizeof(Etudiant),1,outfile) == 1)
     printf("\n%s  %s  %d  %d",e.nom,e.prenom,e.code,e.age);
// re-écriture dans la même fichier du dernier élément lu
  strcpy(e.prenom,"Farida");
  fwrite(&e,sizeof(Etudiant),1,outfile);
  fclose(outfile);
}
else
return 1; //En cas d'échec d'ouverture du fichier
}
```

Ainsi le résultat devient :

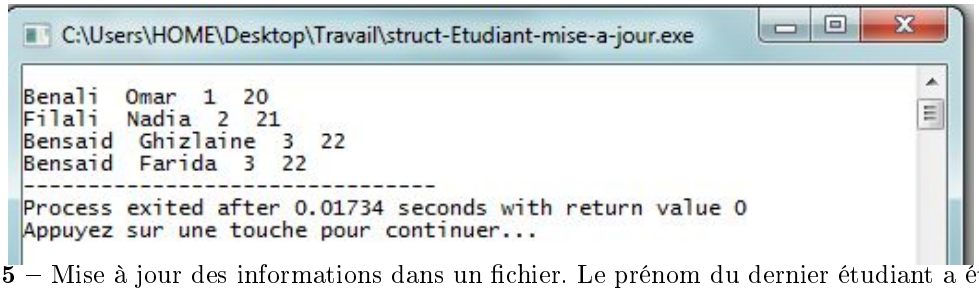


FIGURE 2.25 – Mise à jour des informations dans un fichier. Le prénom du dernier étudiant a été recopié sur le fichier avec le prénom modifié.

Pour modifier le prénom du deuxième étudiant, il faut se positionner sur le deuxième élément et le modifier. Il n'est pas nécessaire de lire tous les éléments pour procéder. D'où l'utilité des fonctions de positionnement (déplacement) pour accéder directement à l'information sollicitée. Ainsi, cet exemple sera traité après l'introduction de ces fonctions dans la section qui suit.

2.4.5 Déplacement dans un fichier (accès direct)

Comme l'accès à l'information stockée dans un fichier est géré à l'aide d'un curseur virtuel indiquant la position dans la fichier pour lire/écrire à une position précise, il est possible d'en disposer pour se positionner sur le fichier. Ainsi, l'accès n'est pas que séquentiel, il peut être aussi direct à l'information sollicitée grâce aux trois fonctions que fournit le langage C pour se positionner dans un fichier :

- **ftell** : indique la position courante du curseur dans le fichier par rapport au début.
  Prototype de la fonction :

  long **ftell**(FILE *fichier) ;

  Où :
  Le nombre renvoyé par la fonction indique la position du curseur dans le fichier.
- **fseek** : positionne le curseur à une position spécifiée. La fonction permet de déplacer le curseur d'un certain nombre de caractères (dans le cas d'un fichier texte) ou de nombres d'octets alloués à un élément (dans le cas d'un fichier binaire) à partir de la position indiquée.
  Prototype de la fonction :

int **fseek**(FILE *fichier, long deplacement, int origine) ;

Où :
deplacement : indique le nombre de caractères (fichier texte) voire le nombre d'octets alloués à un élément (fichier binaire) pour déplacer le curseur dans la fichier. Il peut être un nombre positif (déplacement en avant), nul (= 0) ou négatif (déplacement en arrière).
origine : indique la position à partir de laquelle commence le déplacement. Cette valeur peut être spécifiée par l'une des trois constantes suivantes :
- **SEEK_SET** : indique le début du fichier.
- **SEEK_CUR** : indique la position courante du curseur.
- **SEEK_END** : indique la fin du fichier. ☞ Exemple1 : quelques exemples de déplacement.

```
#include <stdio.h>
int main(){
File *fichier;
fichier = fopen("TEXT.txt","r");
.....................
fseek(fichier,0,SEEK_SET) : // Met le curseur au début du fichier
fseek(fichier,3,SEEK_SET); // Déplace le curseur de 3 caractères après le début
fseek(fichier,-2,SEEK_CUR);// Déplace le curseur de 2 caractères avant la position courante
fseek(fichier,0,SEEK_END);// Déplace le curseur à la fin du fichier
.....................
}
```

☹ La fonction **fseek** est surtout adaptée aux fichiers binaires. Elle peut être utilisée sur des fichiers textes pour uniquement se positionner à la fin ou revenir au début car elle présente certains dysfonctionnements au-delàs de ces déplacements.

☞ Exemple1 : Reprenons Exemple5 de la structure Etudiant où on devait changer le prénom du deuxième étudiant dans le fichier ouvert en mode mise à jour.

```
#include <stdio.h>
#include <string.h>
typedef struct{
   char nom[20],prenom[40];
   int code,age;
}Etudiant;
int main(){
Etudiant e;
char nouv_prenom[40];
FILE *outfile,*infile;
outfile = fopen("etudiant","rb+");
if(outfile != NULL)
{
  fseek (outfile ,1*sizeof(Etudiant), SEEK_SET);
  printf("Mise a jour de l'etudiant :");
  if(fread(&e,sizeof(Etudiant),1,outfile) == 1)
```

```
   {
   printf("\n%s  %s  %d  %d",e.nom,e.prenom,e.code,e.age);
   printf("\n  Entrer le nouveau prenom :");
   gets(nouv_prenom);
   strcpy(e.prenom,nouv_prenom);
   fseek (outfile ,-1*sizeof(Etudiant), SEEK_CUR);
   fwrite(&e,sizeof(Etudiant),1,outfile);
   fclose(outfile);
}}
else
return 1; //En cas d'échec d'ouverture du fichier
// Affichage du contenu du nouveau fichier
infile = fopen("etudiant","r");
if(infile != NULL)
{
printf("\nNouvelle liste des etudiants :") ;
while(fread(&e,sizeof(Etudiant),1,infile) == 1)
  printf("\n%s  %s  %d  %d",e.nom,e.prenom,e.code,e.age);
fclose(infile);
}
else
perror("Impossible d'ouvrir le fichier en lecture!");
}
```

Résultat de l'exécution :

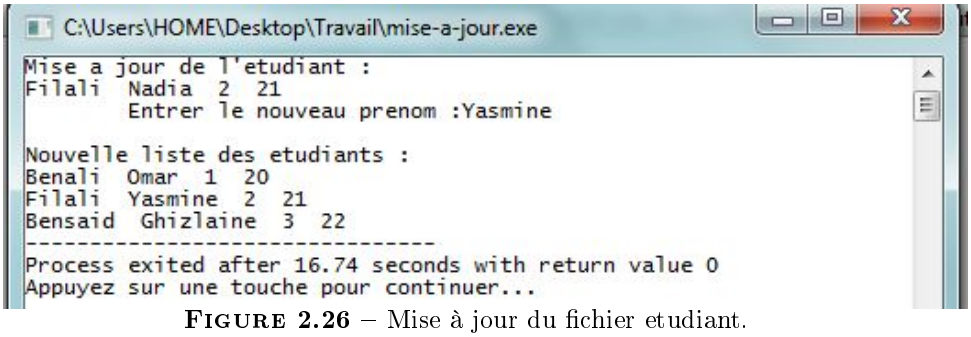


Figure 2.26 – Mise à jour du fichier etudiant.

☞ Exemple2 : déplacements dans un fichier binaire.

```
#include <stdio.h>
#define NB 50
#define SORTIE "fsortie"
int main(void)
{
FILE *infile, *outfile;
int i;
outfile = fopen(SORTIE, "wb");
if (outfile == NULL)
{
fprintf(stderr, "\nImpossible d'ecrire dans le fichier %s\n",SORTIE);
return 1;
}
else
{
for (i =0;i<=NB;i+=4) // Le pas 4 correspond à sizeof(int)=4 octets
/* Ecriture du tableau dans SORTIE */
```

```
fwrite(&i, sizeof(int), 1, outfile);
fclose(outfile);
}
/* Lecture dans SORTIE */
infile = fopen(SORTIE, "rb");
if (infile == NULL)
{
fprintf(stderr, "\nImpossible de lire dans le fichier %s\n",SORTIE);
return 1;
}
else
{
/* Allons à la fin du fichier */
fseek(infile, 0, SEEK_END);
printf("\nPosition actuelle %ld", ftell(infile));
if(fread(&i, sizeof(i), 1, infile)==1)
 printf("\tValeur  actuelle  de i = %d", i);
else
  printf("\nOups rien a lire!");
/* Revenons au début du fichier */
rewind(infile); // Voir la définition et l'emploi de la fonction rewind dans ce qui suit
printf("\nPosition actuelle %ld", ftell(infile));
if(fread(&i, sizeof(i), 1, infile)==1)
printf("\tValeur actuelle  de i = %d", i);
else
  printf("\nOups rien a lire!");
/* Déplacement de 4 int en avant */
fseek(infile, 4 * sizeof(int), SEEK_CUR);
printf("\nPosition actuelle %ld", ftell(infile));
if(fread(&i, sizeof(i), 1, infile)==1)
 printf("\tValeur  actuelle  de i = %d", i);
else
printf("\nOups rien a lire!");
fclose(infile);
return 0;
}}
```

Résultat de l'exécution :

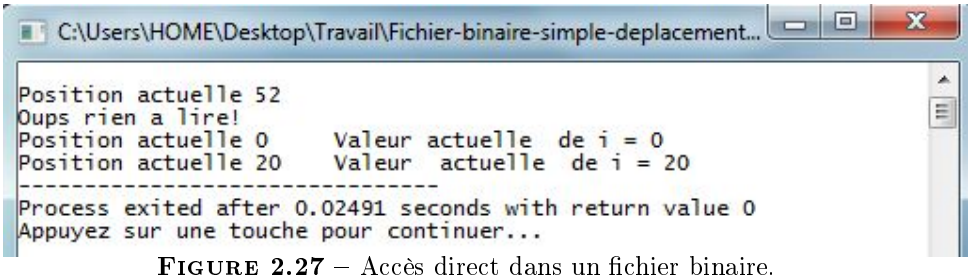


Figure 2.27 – Accès direct dans un fichier binaire.

☞ Exemple3 : déplacements dans un fichier structuré (binaire).

```
#include <stdio.h>
typedef struct {
   char nom[20],prenom[40];
   int code,age;
}Etudiant;
```

```
int main(){
   Etudiant e;
   FILE *infile;
   infile = fopen("etudiant","rb");
  if(infile != NULL)
{
fseek(infile, 0, SEEK_END);
printf("\nPosition actuelle %ld", ftell(infile));
if(fread(&e,sizeof(Etudiant),1,infile) == 1)
  printf("\n%s  %s  %d  %d",e.nom,e.prenom,e.code,e.age);
  else
  printf("\nOups rien a lire!");
/* Revenons au debut du fichier */
rewind(infile);
printf("\nPosition actuelle %ld", ftell(infile));
if(fread(&e,sizeof(Etudiant),1,infile) == 1)
   printf("\n%s  %s  %d  %d",e.nom,e.prenom,e.code,e.age);
   else
  printf("\nOups rien à lire!");
/* Déplacement  en avant */
fseek(infile, 1*sizeof(Etudiant), SEEK_CUR);
printf("\nPosition %ld", ftell(infile));
if(fread(&e,sizeof(Etudiant),1,infile) == 1)
  printf("\n%s  %s  %d  %d",e.nom,e.prenom,e.code,e.age);
  else
  printf("\nOups rien à lire!");
fclose(infile);
}
else
return 1; //En cas d'échec d'ouverture du fichier
}
```

Résultat de l'exécution :

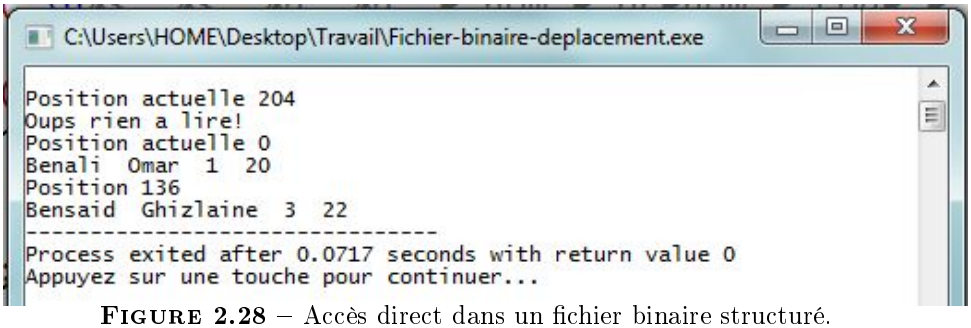


Figure 2.28 – Accès direct dans un fichier binaire structuré.

☺ Il est possible de sauvegarder la position courante du curseur dans un fichier en utilisant la fonction **fgetpos** dont le prototype est :

int **fgetpos**(FILE *fichier, fpos_t *ptr_pos) ;

Où ptr_pos est un pointeur indiquant la position courante du curseur. Cette position peut être récupérée ensuite à l'aide de la fonction **fsetpos** dont le prototype est :

int **fsetpos**(FILE *fichier, const fpos_t *ptr_pos) ;

En cas de succès, c'est la valeur 0 qui est retournée pour les deux fonctions. Le type fpos_t est spécifique à ces deux fonctions et la valeur ptr_pos ne peut être manipulée que par ces deux fonctions.

☞ Exemple : utilisation des deux fonctions **fgetpos** et **fsetpos** sur le fichier etudiant créé précédemment.

```
#include <stdio.h>
#include <string.h>
typedef struct {
   char nom[20],prenom[40];
   int code,age;
}Etudiant;
int main(){
Etudiant e;
fpos_t cle; // Type pointeur spécifique aux 2 fonctions
FILE *infile; // Fichier binaire ouvert en lecture
infile = fopen("etudiant","rb");
if(infile != NULL)
{
 if(fread(&e,sizeof(Etudiant),1,infile) == 1) //Une seule lecture
 printf("\n%s  %s  %d  %d",e.nom,e.prenom,e.code,e.age);
 if(fgetpos(infile,&cle)!=0) // Sauvegarde de la position courante
    {
    perror("Erreur au niveau de fgetpos\n");
   return 1;
    }
printf("\nReprise de la lecture");
if(fsetpos(infile,&cle)!=0) // Reprise à la position sauvegardée
    {
   perror("Erreur au niveau de fsetpos\n");
   return 2;
    }
if(fread(&e,sizeof(Etudiant),1,infile) == 1) // Une seule lecture
  printf("\n%s  %s  %d  %d",e.nom,e.prenom,e.code,e.age);
fclose(infile);
}
else
perror("\n Impossible d'ouvrir le fichier en lecture!");
}
```

Résultat de l'exécution :

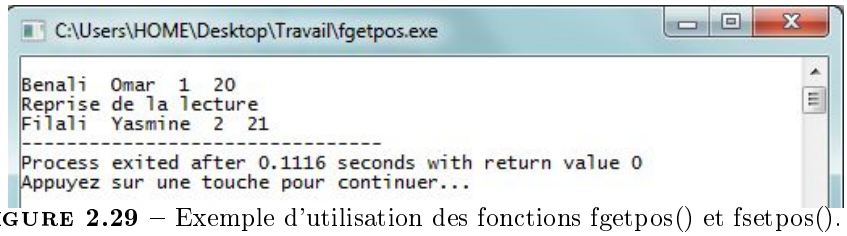


FIGURE 2.29 – Exemple d'utilisation des fonctions fgetpos() et fsetpos().

Attention

Les deux fonctions **fgetpos** et **fsetpos** ont pour deuxième argument l'adresse en mémoire de la valeur de la position courante du curseur et non pas la valeur elle même.

❀ **rewind()** : remet le curseur au début du fichier.

Prototype de la fonction :

int **rewind**(FILE *fichier) ;

Elle est équivalente à :

```
fseek(fichier, 0, SEEK_SET);// Positionnement au début du fichier
clearerr(fichier);// Remettre à zéro l'indicateur d'erreur
```

☞ Exemple : il est possible de réduire le code de Exemple1 en utilisant la fonction **rewind** comme suit :

```
#include <stdio.h>
#include <string.h>
typedef struct {
   char nom[20],prenom[40];
   int code,age;
}Etudiant;
int main(){
Etudiant e;
char nouv_prenom[40];
FILE *outfile;
outfile = fopen("etudiant","rb+");
if(outfile != NULL)
{ fseek (outfile ,1*sizeof(Etudiant), SEEK_SET);
  printf("Mise a jour de l'etudiant :");
  if(fread(&e,sizeof(Etudiant),1,outfile) == 1)
   {printf("\n%s  %s  %d  %d",e.nom,e.prenom,e.code,e.age);
   printf("\n  Entrer le nouveau prenom :");
   gets(nouv_prenom);
   strcpy(e.prenom,nouv_prenom);
   fseek (outfile ,-1*sizeof(Etudiant), SEEK_CUR);
   fwrite(&e,sizeof(Etudiant),1,outfile);
   rewind(outfile);
   printf("\nNouvelle liste des etudiants :") ;
   while(fread(&e,sizeof(Etudiant),1,outfile) == 1)
     printf("\n%s  %s  %d  %d",e.nom,e.prenom,e.code,e.age);
   fclose(outfile);}}
else return 1;  //En cas d'échec d'ouverture du fichier
}
```

Attention

L'utilisation du mode mise à jour (lecture et écriture) sur un fichier binaire nécessite certaines précautions qui sont les suivantes :

- ☹ En cas de lecture juste après une opération d'écriture dans le fichier, il faut tout d'abord utiliser la fonction **fflush** ou une fonction de positionnement.
- ☹ En cas d'écriture juste après une opération de lecture dans le fichier si la fin de fichier n'est pas atteinte, il faut tout d'abord utiliser une fonction de positionnement.

2.4.6 Renommer et supprimer un fichier

Le langage C permet de renommer des fichiers et les supprimer s'ils sont existants à l'aide des fonctions suivantes :

- **rename** : renomme un fichier, elle retourne la valeur 0 en cas de succès.
  Prototype de la fonction :

  int **rename**(const char *nom_physique1, const char *nom_physique2) ;

  Où :
  nom_physique1 : désigne le nom physique du fichier à renommer.
  nom_physique2 : désigne le nouveau nom physique du fichier.
  ☞ Exemple :

```
rename("test.txt,"nouveau.txt"); // Le fichier test.txt est renommé en nouveau.txt
```

- **remove** : supprime un fichier existant, elle opère sans demander de confirmation et le fichier n'est pas mis dans la corbeille.

  int **remove**(const char *nom_physique) ;

  ☞ Exemple :

```
remove("nouveau.txt"); // Supprime fichier nouveau.txt du disque
```

2.5 Résumé

Ce chapitre a introduit la notion de fichier et présenté les principales fonctions permettant d'opérer sur les fichiers en langage C selon leur type (texte ou binaire) et le mode d'accès sollicité (séquentiel ou direct). Plusieurs exemples relatifs à chaque fonction introduite ont été explicités avec détails. Les codes source ont été présentés avec capture d'écran des résultats de leur exécution. Ce chapitre sera complété par quelques énoncés d'exercices résolus et d'autres proposés comme test.

Exercice 3 *Compter le nombre de caractères dans un fichier texte.*

```
#include <stdio.h>
int main(){
char nom_fich1[]="essai1.txt";
char c;
int nbcar=0;
FILE *infile;
infile = fopen(nom_fich1,"r");
if(infile == NULL)
 {
  printf("Impossible d'ouvrir le fichier %s en lecture!\n",nom_fich1);
  return 1;
 }
 while(!feof(infile))
 { c = fgetc(infile);
   if(c != EOF && c!= '\n')
         nbcar++;
}
fclose(infile);
printf("\n Nombre de caractères du fichier = %d\n",nbcar);
}
```

Contenu du fichier essai1.txt :

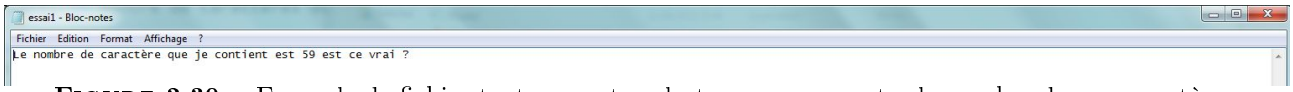


FIGURE 2.30 – Exemple de fichier texte ouvert en lecture pour compter le nombre de ses caractères.

Résultat de l'exécution :

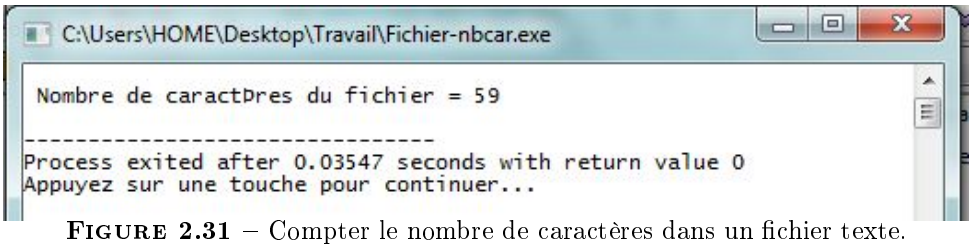


FIGURE 2.31 – Compter le nombre de caractères dans un fichier texte.

Exercice 4 *Copie du contenu d'un fichier essai1.dat dans un fichier essai2.dat.*

```
#include <stdio.h>
int main()
{char nom_fich1[]="essai1.dat";
 char nom_fich2[]="essai2.dat";
 char c;
 FILE *infile, *outfile;
 infile = fopen(nom_fich1,"r");
 if(infile == NULL)
 {
  printf("Impossible d'ouvrir le fichier %s en lecture!\n",nom_fich1);
 return 1;
 }
 outfile = fopen(nom_fich2,"w");
 if(outfile == NULL)
 {
```

```
  printf("Impossible d'ouvrir le fichier %s en écriture!\n",nom_fich2);
 return 2;
 }
 while(!feof(infile))
 { c = fgetc(infile);
   fputc(c,outfile);
}
fclose(infile);
fclose(outfile);
}
```

Contenu du fichier essai1.dat :

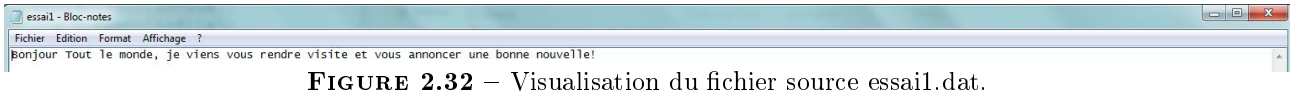


FIGURE 2.32 – Visualisation du fichier source essai1.dat.

Contenu du fichier résultat essai2.dat après l'exécution du code :

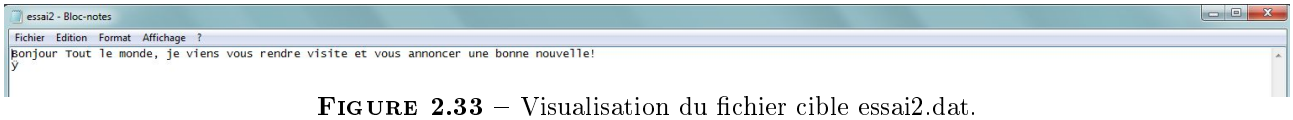


FIGURE 2.33 – Visualisation du fichier cible essai2.dat.

Exercice 5 *: Fichier structuré*

Soit T un tableau d'au plus 100 étudiants. En considérant la structure Etudiant définie par :
Matricule : entier ;
Nom, Prenom : chaine [20] ;
Moyenne : réel ;

a) Écrire le code en C permettant de recopier tous les étudiants admis appartenant à T dans un fichier ADMIS de type Etudiant (un étudiant est admis si sa moyenne est supérieure ou égale à 10).

b) Écrire le code permettant d'afficher l'étudiant ayant la meilleure moyenne.

c) En supposant le fichier trié par ordre décroissant des moyennes (de la plus grande à la plus petite), donner le code affichant le dernier et l'avant dernier étudiant dans la liste.

```
#include <stdio.h>
#define NB 6 // Nombre d'étudiants à saisir
//Déclaration de la structure
typedef struct{
   int Matricule;
   char Nom[20],Prenom[20];
   float Moyenne;}Etudiant;
Etudiant T[10],e;// e est utilisée pour la saisie des info d'un étudiant
// Recopier les étudiants admis
/* Ici la solution est donnée en utilisant un fichier binaire, bien entendu il est possible d'utiliser
     un fichier texte et lire/écrire dedans à
l'aide de fscanf() et fprintf() (lecture/écriture formatée) */
void recopier(){
FILE *outfile;
int i;
outfile = fopen("ADMIS","wb");
```

```
if(outfile==NULL)
  printf("Impossible d'ouvrir le fichier en écriture \n");
  else
  {
  for(i=0;i<NB;i++)
     if(T[i].Moyenne >= 10)
       {
        printf("%s\t%s\n",T[i].Nom,T[i].Prenom);
        fwrite(T+i,sizeof(Etudiant),1,outfile);
       }
  fclose(outfile);
}}
//*******************************************************
// Afficher le ou les étudiants ayant la meilleure moyenne
void bestMoy(float max) // max est calculée dans le main
{FILE *outfile;
outfile = fopen("ADMIS","rb");
if(outfile==NULL)
  printf("Impossible d'ouvrir le fichier en lecture \n");
else
{
   printf("\nEtudiants dont la moyenne est %f :",max);
  while(!feof(outfile))
{
 fread(&e,sizeof(Etudiant),1,outfile);
 if(e.Moyenne == max)
         printf("\nMatricule : %d\tNom : %s\tPrenom : %s\n", e.Matricule,e.Nom,e.Prenom);
}
fclose(outfile);
}}
//*******************************************************
// Afficher le dernier et avant dernier étudiant
void derniers()
{FILE *outfile;
outfile = fopen("ADMIS","rb");
if(outfile==NULL)
  printf("Impossible d'ouvrir le fichier en lecture \n");
 else
 {
printf("\nDerniers etudiants figurant dans le fichier des admis:\n");
fseek(outfile,0,SEEK_END);
fread(&e,sizeof(Etudiant),1,outfile);
 printf("\nDernier -> Matricule : %d\tNom : %s\tPrenom : %s\n", e.Matricule,e.Nom,e.Prenom);
fseek(outfile,-2*sizeof(Etudiant),SEEK_CUR);
fread(&e,sizeof(Etudiant),1,outfile);
 printf("\nAvant dernier -> Matricule : %d\tNom : %s\tPrenom : %s\n", e.Matricule,e.Nom,e.Prenom);
}}
int main(){
FILE *outfile;
float max=0;
int i;
//Boucle pour la saisie des infos du Tableau pour chaque étudiant
printf("Saisie des informations des etudiants \n");
for(i=0;i<NB;i++)
{
printf("Entrer le matricule :");
```

```
scanf("%d",&T[i].Matricule);
printf("Entrer le nom :");
scanf("%s",T[i].Nom);
printf("Entrer le prenom :");
scanf("%s",T[i].Prenom);
printf("Entrer la moyenne :");
scanf("%f",&T[i].Moyenne);
}
// Recopier dans un fichier les etudiants admis
recopier();
//Afficher l'etudiant ayant la meilleure moyenne
/* Il est aussi possible de chercher le max directement dans le tableau au lieu du fichier*/
outfile = fopen("ADMIS","rb");
if(outfile==NULL)
  {printf("Impossible d'ouvrir le fichier en lecture \n");
   return 1;
  }
while(!feof(outfile))
{
fread(&e,sizeof(Etudiant),1,outfile);
 if(e.Moyenne > max)
          max = e.Moyenne ;
}
bestMoy(max);
//Afficher le dernier et avant dernier étudiants
derniers();
}
```

Résultat de l'exécution :

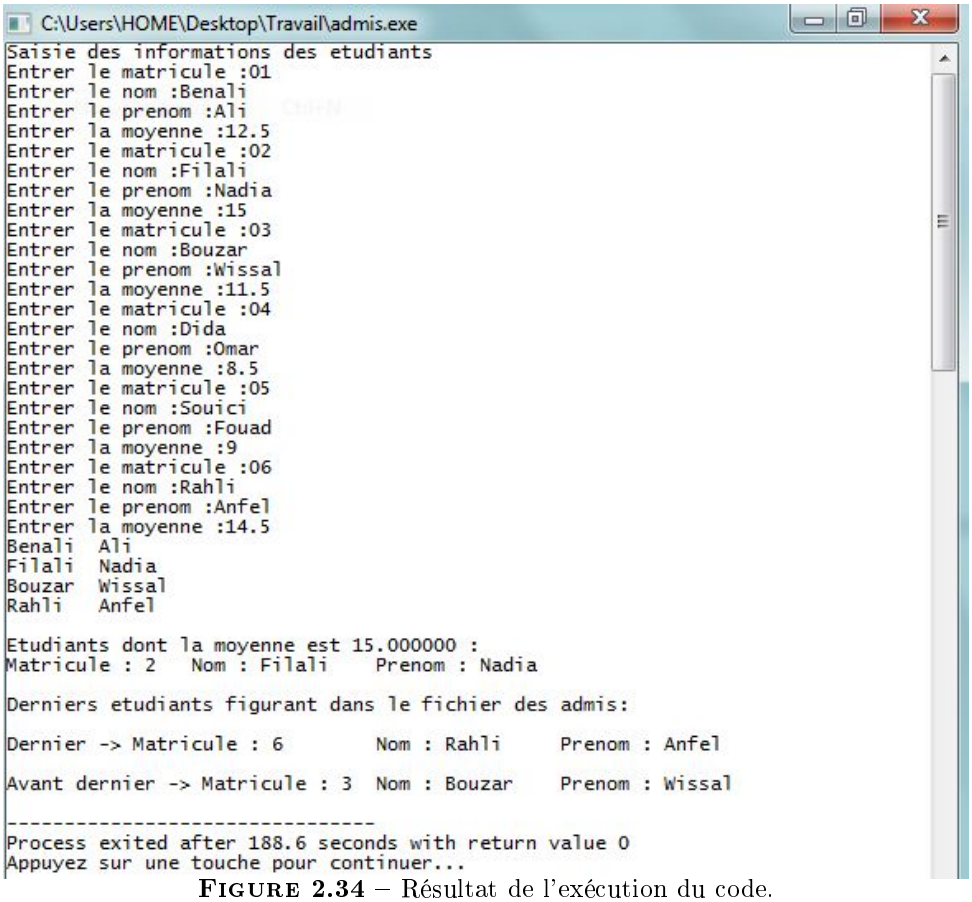


Figure 2.34 – Résultat de l'exécution du code.

Exercice 6 *Compléter ce code en ajoutant une fonction affichant le nombre d'enseignantes et en prévoyant une mise à jour des informations saisies.*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define nb 10
typedef struct{
   int num;
   float salaire;
   char nom[20],prenom[40];
   char sexe;  // Le caractère 'F' pour féminin et 'M' pour masculin
   char dept[20];
   int fonction;  // 0 pour employé  et 1 pour enseignant
}Employe;
int main(){
int choix;
Employe e[nb],v[nb];
FILE *infile, *outfile;
int i;
do
{
printf("\n\nMenu :\n");
printf("1 : Saisie des employes\n");
printf("2: Affichage de tous les employes\n");
printf("3 : Quitter\n");
printf("Faites votre choix S.V.P : ");
scanf("%d",&choix);
switch(choix)
 {case 1:
   outfile = fopen("employe","wb");
   if(outfile !=NULL)
   {
   for(i=0;i<nb;i++)
    {printf("\n      Saisir l'employe N° %d \n",i+1);
     printf("\nEntrer le numero : ");
     scanf("%d",&e[i].num);
     printf("\nEntrer le salaire : ");
     scanf("%f",&e[i].salaire);
     printf("\nEntrer le nom : ");
     scanf("%s",e[i].nom);
     printf("\nEntrer le prenom : ");
     scanf("%s",e[i].prenom);
     fflush(stdin);
     printf("\nEntrer le sexe F/M: ");
     scanf("%c",&e[i].sexe);
     printf("\nEntrer le departement : ");
     scanf("%s",e[i].dept);
     printf("\nEntrer la fonction 0/1 : ");
     scanf("%d",&e[i].fonction);
     }
     fwrite(e,sizeof(Employe),nb,outfile);
     fclose(outfile);
      }
     break;
```

```
case 2 :
   // Affichage de tous les employes saisis
   infile = fopen("employe","rb");
  if(infile != NULL)
   {
   fread(v,sizeof(Employe),nb,infile);
   for(i=0;i<nb;i++)
    printf("\nEmploye N°%d : %d  %f %s  %s  %c  %s  %d",i+1,v[i].num,v[i].salaire,v[i].nom,v[i].prenom
     ,v[i].sexe,v[i].dept,v[i].fonction);
 fclose(infile);
}
break;
case 3 :
 // Autre traitement à compléter  ou Fin
    printf("\n Fin de traitement, Au revoir et  Merci !\n");
    break;
default  :
   printf(" Oups Vous avez fait un mauvais choix !\n");
}} while(choix!=3);
}
```

Exercice 7 : ***Traitement sur un fichier structuré à faire***

Etant donné une structure Voiture d'une agence de location de voitures. La structure est définie par :

- ***Le matricule*** *qui est lui même une structure définie par :*
  - *char nb[7] ;*
  - *int annee ;*
  - *int Wilaya ;*
- ***La marque*** *(chaîne de 10 caractères) ;*
- ***L'état*** *du véhicule (0 :disponible et 1 : loué) ;*

Ecrire la fonction principale (main) permettant de :

a) Créer un fichier binaire permettant d'enregistrer les informations de 20 véhicules de l'agence.

b) Afficher le modèle et la marque de tous les véhicules disponibles (non loués) de l'année en cours.

c) Afficher le nombre de véhicules par wilaya.

d) Afficher l'état du dernier véhicule dans le fichier.

CHAPITRE 3

LES LISTE CHAÎNÉES

Sommaire

Figures

3.1 Introduction

Rappelons tout d'abord ce que veut dire la notion de structure de données. Une structure de données est une manière d'organiser les données utilisées dans un algorithme pour l'implémenter sur machine. On distingue les structure de données séquentielles ou linéaires, les arbres et les graphes. Cette partie introduit les structures linéaires de base, l'appellation est due à la notion de séquence qui existe entre les éléments de la structure. Jusqu'ici, nous avons vu les tableaux dont les éléments de même type sont organisés de façon contiguë en mémoire et dont la réservation en mémoire nécessite à priori, le nombre maximum d'éléments (la taille maximale) qui désormais ne pourra pas changer mais il y'a aussi les listes, les piles et les files. Les trois derniers modèles sont considérés comme des types abstraits de données puisqu'ils peuvent concrètement être implémentés soit sous forme de tableaux (structures de données statiques), soit sous forme de listes chaînées (structures de données dynamiques) qui contrairement aux tableaux, elles n'ont pas de dimension figée à la création et leurs éléments sont dispersés en mémoire et reliés entre eux par des références ou pointeurs. Les éléments d'une structure de données dynamique sont crées au fur et à mesure (au besoin) par allocation dynamique de la mémoire et peuvent être détruites à tout moment comme nous allons voir avec plus de détails dans ce qui suit. Avant cela, il convient de rappeler la notion de pointeurs et son rapport avec l'allocation dynamique de la mémoire, principe de base des listes chaînées.

3.2 Les pointeurs

Le type pointeur à été déjà abordé et utilisé dans le passage des paramètres par référence dans les sous-programmes. Néanmoins, un retour sur les pointeurs à ce niveau est nécessaire pour souligner certaines propriétés primordiales quant à leur utilisation dans les listes chaînées. Rappelons tout d'abord que le type pointeur est un type de données dont la valeur fait référence directement à une donnée (variable) qui réside quelque part en mémoire à une certaine adresse. Ainsi, toute donnée en mémoire centrale est caractérisée par un contenu (sa valeur) et une adresse.

❀ Opérateurs « * » et « & » :

Un pointeur est donc une variable contenant une adresse mémoire pointant vers une donnée. Cette adresse peut être obtenue à l'aide de l'opérateur « & », ainsi si a est une donnée, $\&a$ est son adresse en mémoire. Si p est un pointeur, $*p$ représente la valeur se trouvant à l'adresse pointée par p.

☞ Exemple1 :

Figure 3.1 – Pointeurs : adresse et contenu.

```
int a = 5, b, *p;
p = &a; // p est l'adresse en mémoire de la variable a
b = *p; // b est la valeur stockée dans la case d'adresse p qui vaut donc 5
```

☞ Exemple2 :

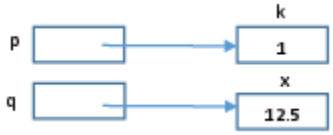


Figure 3.2 – Pointeurs : accès au contenu des variables pointées.

```c
#include <stdio.h>
main(){
   int k=1,*p;
   float x=12.5,*q;
   p = &k; // p est l'adresse de k en mémoire
   q = &x; // q est l'adresse de x en mémoire
   printf("    *p = %d\n   *q = %.2f\n",*p,*q);
   *p = 3; *q=0; /* Le changement des valeurs pointées par p et q est en fait le changement des
    valeurs de k et x respectivement */
   printf("    *p = %d\n   *q = %.2f\n",*p,*q);
}
```

Exécution du code :

Figure 3.3 – Exemple de manipulation de pointeurs.

Remarque *: les opérateurs « & » et « * » ont des effets symétriques :* $*(\&a) = a$.

❀ Pointeur NULL :

Un pointeur peut ne pointer sur aucune variable, c'est à dire sur rien. Dans ce cas on écrit :

```c
int *p ;
p = NULL ; // Ou bien p = 0;
```

❀ Type et variables pointeurs :

Il est possible d'associer à tout type **t** (type de base) un nouveau type de pointeurs sur des données du type **t**. L'adresse d'une donnée de type t est désignée par $t*$, par exemple une adresse d'entiers est désignée par int^*, une adresse de réels est désignée par **float***, ...etc.

❀ Pointeur sur un pointeur :

Comme un pointeur est lui aussi une variable. Son adresse en mémoire peut être aussi indiquée par un pointeur comme dans l'exemple suivant :

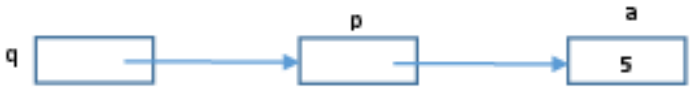


Figure 3.4 – Pointeurs : pointeur sur un pointeur.

```c
#include<stdio.h>
main(){
   int a = 5, *p, **q; // q est un pointeur sur un pointeur p
   p=&a; // p est l'adresse en mémoire de la variable a
   q=&p; // q est l'adresse en mémoire de la variable p
   printf("\np = %d\t *p = %d\t q = %d\t *q = %d\t *(*q) = %d\n",p,*p,q,*q,*(*q));
}
```

Exécution du code :

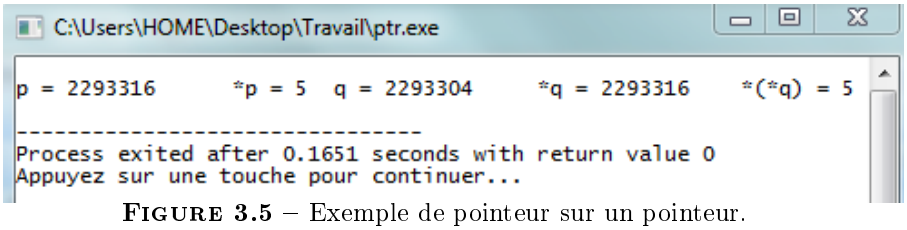


FIGURE 3.5 – Exemple de pointeur sur un pointeur.

Observer que les déclarations des deux pointeurs p et q dans le code diffèrent. Le pointeur p est déclaré par int *p qui indique que p pointe sur une variable de type int alors que le pointeur q est déclaré par int **q qui indique que le pointeur q pointe sur un pointeur de int.

❀ Pointeur sur une structure :

Les structures ont déjà été abordées et on a vu comment les variables de type structure sont déclarées. Ici, il est utile de rappeler qu'il est aussi possible d'utiliser des pointeurs sur des structures.

☞ Exemple :

```
struct point{
int  x,y;
};
struct point *p ; // p est un pointeur sur une structure de type point
// Pour accéder au champ de la variable pointée par p on écrit :
(*p).x = 12.5;
// Ou bien
p->x = 12.5
```

Remarque *: dans la première notation, les parenthèses sont obligatoires puisque l'opérateur « . » est plus prioritaire que l'opérateur « * ». Cette notation a été simplifiée par l'introduction de l'opérateur « -> ». Ainsi, pour accéder aux membres (champs) d'une variable de type structure pointée par un pointeur, les deux expressions suivantes sont équivalentes :*

*(*pointeur).champ*
pointeur->champ

La notion de pointeur est étroitement liée à la gestion dynamique de l'espace mémoire occupé par les variables d'un programme qui peut être obtenu de manière statique ou de manière dynamique.

3.3 Gestion dynamique de la mémoire

L'allocation dynamique de la mémoire permet la réservation de l'espace mémoire pour les données du programme au moment de son exécution non lors de la compilation contrairement à l'allocation statique, où l'espace mémoire est réservé à l'avance sans connaissance à priori de l'espace réellement occupé et ne peut être modifié. Le langage C fournit des primitives pour gérer dynamiquement la mémoire comme désigner, créer et libérer de l'espace mémoire.

Allocation de la mémoire : pour allouer de la mémoire, deux fonctions sont disponibles en C, implémentées dans la librairie <**stdlib.h**> :

- La fonction **malloc** : est utilisée pour allouer de l'espace mémoire comme suit :

  Type *identificateur = (Type*) **malloc**(taille) ;

  Où
  identificateur : représente le nom de la variable pointeur vers l'espace alloué.
  Type : le type de la variable.
  taille : le nombre d'octets pour représenter le type indiqué par Type. Le nombre d'octets peut varier en fonction du compilateur.
  La valeur retournée est un pointeur sur la zone mémoire allouée, ou **NULL** en cas d'échec de l'allocation. Dans le cas où on ignore à priori le nombre d'octets alloué à chaque type, il suffit d'utiliser la fonction **sizeof** comme suit :

  Type *identificateur = (Type*) **malloc**(nbre_elements × **sizeof**(Type)) ;

  Où :
  nbre_elements : est le nombre d'éléments de la variable dont le nom est donné par identificateur.

  ☞ Exemple1 :

```
#include <stdio.h>
#include<stdlib.h>
main(){
int *tab = malloc (30*sizeof(int)); // Réservation d'un tableau d'entiers tab de 30 cases
if(tab==NULL)
{printf("Allocation impossible");
 exit(EXIT_FAILURE); /*Fin anormale indiquée par la constante EXIT_FAILURE disponible dans <
    stdlib.h>, tout comme EXIT_SUCCESS */
}}
}
```

  ☞ Exemple2 :

```
float *Tab = malloc(20 * sizeof(float));// Réservation d'un tableau de réels Tab de 30 cases
```

- La fonction **calloc** : permet également d'allouer de l'espace mémoire avec en plus l'initialisation des zones mémoires allouées à 0 (tous les bits de la zone mémoire allouée sont mis à 0). Son utilisation est similaire à celle de **malloc** avec une syntaxe légèrement différente :

  Type *identificateur = (Type*) **calloc**(nbre_elements,**sizeof**(Type)) ;

  La fonction retourne l'adresse de la zone mémoire allouée selon la taille indiquée par *nbre_elements* initialisée à 0 ou **NULL** en cas d'échec.

  ☞ Exemple :

```
---------------------
float *Tab = calloc(20,sizeof(float));//Les 20 cases mémoires allouées sont initialisées à 0
---------------------
```

❀ La fonction **realloc** : permet de modifier (redimensionner) la taille de la zone mémoire précédemment allouée. Elle est utilisée comme suit :
☞ Exemple :

```
--------------------
float *Tab = calloc(20,sizeof(float));
--------------------
Tab = realloc(Tab,80*sizeof(float)); /* Le tableau précédemment alloué est augmenté de 60
    cases supplémentaires */
--------------------
```

La fonction retourne l'adresse de la zone de la nouvelle taille demandée ou **NULL** en cas d'échec. Les anciennes données qui y sont stockées sont conservées ou tronquées si la taille a diminué.

Libération de la mémoire : en langage C, la libération de l'espace mémoire est explicite. l'espace précédemment alloué par **malloc** ou **calloc** peut être libéré à l'aide de la fonction **free** qui est également disponible dans la librairie <**stdlib.h**>, comme suit :

free(identificateur) ;

Où
identificateur : représente le nom de la variable pointeur indiquant l'adresse de la zone mémoire à supprimer. La zone mémoire libérée est récupérée par le système et la variable pointeur correspondante n'est pas initialisée à **NULL** mais a une adresse indéterminée.
☞ Exemple :

```
float *Tab = malloc(20*sizeof(float));
--------------------
free(Tab); // Libération de la mémoire occupée par Tab
```

☺ L'allocation/désallocation de la mémoire nécessite l'intégration dans le code de la librairie <**stdlib.h**>.
☺ Après l'invocation de la fonction **malloc**, il faut s'assurer que l'allocation s'est bien faite à ce moment là. Auquel cas, prévoir un message d'erreur.
☺ Avant de libérer une zone mémoire, il est recommandé de s'assurer que son pointeur n'est pas **NULL**.
☺ Le langage C++ fournit l'opérateur **new** pour allouer de l'espace et l'opérateur **delete** pour libérer l'espace alloué, tous les deux implémentés dans la librairie <**stdlib.h**> (noter qu'ils sont dits opérateurs plutôt que fonctions car leur emploi ne nécessite pas l'utilisation des parenthèses).

3.4 Les listes chaînées

Une liste linéaire sur un ensemble E est une suite finie $(e_1, e_2, ..., e_n)$ d'éléments de E accessibles soit directement par leurs indices si la liste est implémentée en mémoire centrale de façon contiguë (sous forme de tableau), soit indirectement si la liste est représentée par une suite de cases chaînées où chaque case est allouée dynamiquement (au moment de l'exécution du code) d'où l'appellation liste chaînée. La forme générale d'une telle liste est comme suit :

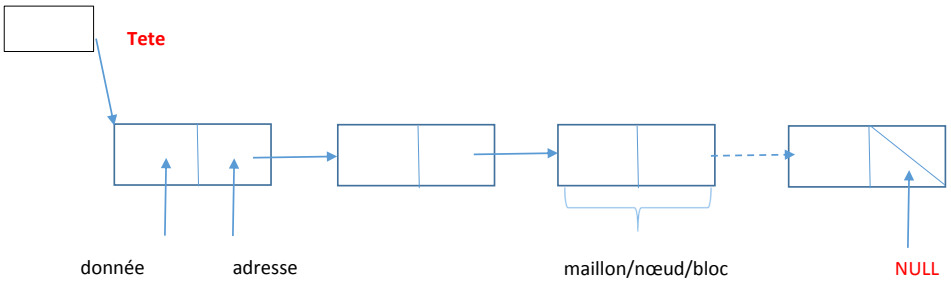


Figure 3.6 – Structure d'une liste chaînée.

Chaque maillon ou nœud constituant la liste chaînée contient 2 champs :

- un champ valeur contenant la donnée proprement dite qui peut être de type simple ou composé (structure).
- un champ adresse (pointeur) contenant l'adresse du maillon contenant l'élément suivant dans la liste. Il peut être égal à la constante **NULL** pour indiquer le fin de la liste.

L'adresse du premier maillon (la tête de la liste) doit toujours être sauvegardée dans une variable pour pouvoir manipuler la liste, ce qui veut dire que la liste est accessible par sa tête de liste . Cette adresse est positionnée à la valeur **NULL** si la liste est vide (ne contient aucun maillon) comme illustré dans la Figure 3.7.

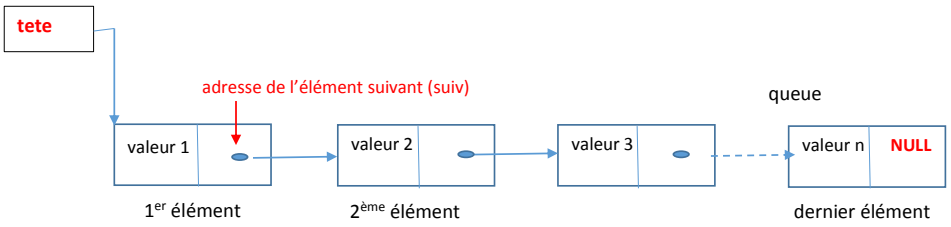


Figure 3.7 – Liste chaînée simple.

Syntaxe de déclaration d'une liste chaînée :

```
struct liste{
int valeur; // Champ donnée
struct liste *suiv; // Champ adresse pointant vers le prochain élément
};
struct liste *tete = NULL; // Initialisation à liste vide
```

Ou bien en utilisant **typedef** :

```
typedef struct liste{
int valeur;
struct liste *suiv;
}Liste;
Liste *tete = NULL; // Initialisation à liste vide
```

Syntaxe d'allocation de la mémoire à un nouvel élément en C :

```
#include<stdlib.h> // Pour utiliser malloc()
struct liste{
int valeur;
struct liste *suiv;
};
struct liste *tete = NULL; // Initialisation à liste vide
// Allocation de la mémoire à un nouvel élément
struct liste *nouv = (struct liste*) malloc(sizeof(struct liste));
```

Ou bien en utilisant **typedef** :

```
#include<stdlib.h> // Pour utiliser malloc()
typedef struct liste{
int valeur;
struct liste *suiv;
}Liste;
Liste *tete = NULL; // Initialisation à liste vide
// Allocation de la mémoire à un nouvel élément
Liste *nouv = (Liste*) malloc(sizeof(Liste));
```

Syntaxe d'allocation de la mémoire à un nouvel élément en C++ :

```
#include<stdlib.h> // Pour utiliser new du C++
struct Liste{
int valeur;
struct Liste *suiv;
};
Liste *tete = NULL; // Initialisation à liste vide
// Allocation de la mémoire à un nouvel élément
Liste *nouv = new Liste;
```

☺ Le langage C++ a apporté une certaine souplesse à la manipulation des structures et des pointeurs. Côté structures, il est possible d'utiliser le nom de la structure dans une déclaration sans une utilisation à priori du mot-clé **typedef** et coté pointeurs, l'utilisation de l'opérateur **new** de la librairie <**stdlib.h**>. Ce qui a permis une écriture de code plus simple.

3.5 Quelques opérations sur les listes chaînées

Décrivons quelques opérations sur les listes chaînées en utilisant la déclaration suivante pour sa simplicité :

```
#include<stdlib.h>
typedef struct liste{
int valeur;
struct liste *suiv;
}Liste;
```

Parmi ces opérations, on peut citer :

Vérifier si une liste existe : cette opération consiste tout simplement à tester si la valeur de la variable pointeur de tête de liste est égale à **NULL**.
En C :

```
int estVide(Liste *tete){
if(tete == NULL)
    return 1;
      else
    return 0;
}
```

En C++, en incluant la librairie <stdbool.h> :

```
#include <stdbool.h>
bool estVide(Liste *tete){
return (tete == NULL);
}
```

Affichage des éléments d'une liste : le code suivant permet d'afficher tous les éléments d'une liste si elle existe.

```
void affichListe(Liste *tete){
Liste *courant = tete; // Le parcours de la liste débute à partir de tete (premier élément)
if(courant==NULL)
```

```
    printf("Rien a afficher, la liste est vide!\n");
    else
    while(courant != NULL) // Parcours de la liste jusqu'à la fin
     {
      printf("%d ", courant->valeur);
      courant=courant->suiv;
     }
printf("\n");
}
```

Ce code est aussi équivalent au code ci-dessous puisque le pointeur tete ne sera modifié que localement, à l'extérieur de la fonction il reste intact (appel par valeur).

```
void affichListe(Liste *tete){
// Le parcours de la liste débute à partir de tete (premier élément)
if(tete==NULL)
    printf("Rien a afficher, la liste est vide!\n");
    else
    while(tete != NULL) // Parcours de la liste jusqu'à la fin
     {
      printf("%d ", tete->valeur);
      tete=tete->suiv;
     }
printf("\n");
}
```

Il est aussi possible d'afficher la liste à partir du dernier élément de la liste (queue de liste) en utilisant la fonction récursive suivante :

```
void afficherSensInverse(struct  liste *tete)
{
  if (tete != NULL)
    {
      afficherSensInverse(tete->suiv);
      printf("%d ", tete->valeur);
    }}
```

Création d'une liste : la création d'une liste peut se faire par une insertion répétée d'un certain nombre de valeurs saisies. Pour se faire, donnons le code complet qui après création de la liste affiche son contenu sur l'écran. Noter qu'à priori, le nombres d'éléments de la liste est inconnu et les éléments sont ajoutés au fur et à mesure selon que l'on veuille continuer ou non. La fonction **creation** se charge de la création de la liste et la fonction **affichListe** affiche son contenu une fois qu'elle est créée.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct liste{
   int valeur; // Champ donnée de l'élément
   struct liste *suiv; // Champ adresse pointant vers l'élément suivant
}Liste;
Liste *tete = NULL;
```

```
Liste *creation(Liste *tete, int x);// La fonction retourne un pointeur de type Liste
void affichListe(Liste *tete);
main(){
int x;
char reponse = 'o'; // Réponse à la question : Voulez vous continuer (oui/non)?
do
 {printf("Entrer la valeur : ");
  scanf("%d", &x);
tete = creation(tete,x); // Actualisation de la valeur de tete
  printf("     Voulez vous continuer o/n ? : ");
  fflush(stdin);// Vider le buffer (mémoire tampon) en cas de non lecture de la réponse
  reponse = getchar();
  } while (reponse == 'o');
printf("\nContenu de la liste :\n");
affichListe(tete);
}
Liste *creation(Liste *tete, int x){
Liste *p =  (Liste*) malloc(sizeof(Liste)); /* Ou bien Liste *p = new Liste tout simplement en
    utilisant l'opérateur new du C++ */
p->valeur = x;
p->suiv = tete;
return p; // Nouvelle tête
}
void affichListe(Liste *tete){
Liste *courant = tete;
 if(courant==NULL)
   printf("Rien a afficher, la liste est vide!\n");
   else
   while(courant != NULL)
     {
      printf("%d  ", courant->valeur);
      courant=courant->suiv;
     }
printf("\n");}
```

Résultat de l'exécution du code :

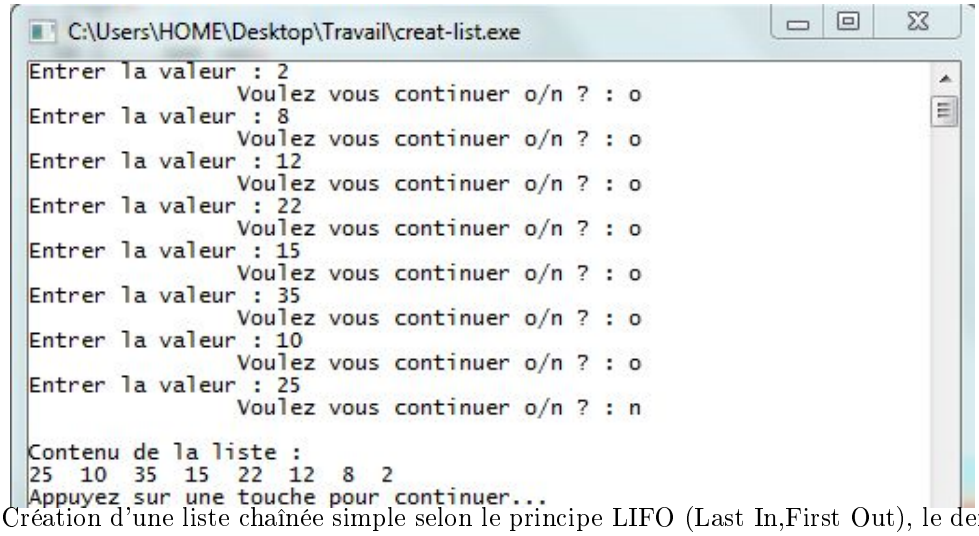


Figure 3.8 – Création d'une liste chaînée simple selon le principe LIFO (Last In,First Out), le dernier entré sera le premier dans la liste.

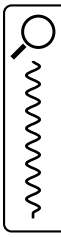
Utilisons maintenant une fonction **creation** de type void, le code devient en C :

```
#include <stdio.h>
#include <stdlib.h>
typedef struct liste{
   int valeur;
   struct liste *suiv;
}Liste;
Liste *tete = NULL;
void creation(Liste **tete, int x);
void affichListe(Liste *tete);
main(){
int x;
char reponse = 'o';
do
 {printf("Entrer la valeur : ");
  scanf("%d", &x);
  creation(&tete,x); // Actualisation de la valeur de tete
  printf("     Voulez vous continuer o/n ? : ");
  fflush(stdin);/* Vider le buffer en cas de non lecture de la réponse */
  reponse = getchar();
  } while (reponse == 'o');
printf("\nContenu de la liste :\n");
affichListe(tete);
}
void creation(Liste **tete, int x) /* Noter que tete est précédée de 2 étoiles l'une indiquant
    que c'est un pointeur et l'autre marquant l'appel par adresse pour récupérer sa nouvelle
    valeur dans le cas d'une éventuelle modification */
{Liste *p =  (Liste*) malloc(sizeof(Liste));
p->valeur = x;
p->suiv = *tete;
*tete = p; // Nouvelle tête
}
void affichListe(Liste *tete){
 Liste *courant = tete;
 if(courant==NULL)
    printf("Rien a afficher, la liste est vide!\n");
    else
    while(courant != NULL)
     {
      printf("%d  ", courant->valeur);
      courant=courant->suiv;
     }
 printf("\n");}
```

Ou bien en C++ :

```
#include <stdio.h>
#include <stdlib.h>
typedef struct liste{
   int valeur;
   struct liste *suiv;
}Liste;
Liste *tete = NULL;
void creation(Liste *&tete, int x);
void affichListe(Liste *tete);
```

```
main(){
int x;
char reponse = 'o';
do
  {printf("Entrer la valeur : ");
   scanf("%d", &x);
   creation(tete,x); // Actualisation de la valeur de tete
   printf("    Voulez vous continuer o/n ? : ");
   fflush(stdin);/* Vider le buffer en cas de non lecture de la réponse */
   reponse = getchar();
   } while (reponse == 'o');
printf("\nContenu de la liste :\n");
affichListe(tete);
}
void creation(Liste *&tete, int x) /* Noter que tete est précédée d'une étoile indiquant que c'
    est un pointeur et l'opérateur & marquant l'appel par adresse pour récupérer sa nouvelle
    valeur dans le cas d'une éventuelle modification */
{Liste *p =  (Liste*) malloc(sizeof(Liste));
p->valeur = x;
p->suiv = tete;
tete = p; // Nouvelle tête
}
void affichListe(Liste *tete){
 Liste *courant = tete;
 if(courant==NULL)
    printf("Rien a afficher, la liste est vide!\n");
    else
    while(courant != NULL)
     {
      printf("%d  ", courant->valeur);
      courant=courant->suiv;
     }
printf("\n");}
```



Observer que dans la Figure 3.8, les valeurs sont affichées selon le principe Last In First Out (LIFO), c'est à dire le dernier entrée sera le premier dans la liste. La liste est donc affichée dans le sens inverse, pour qu'elle soit affichée dans le bon sens, il va falloir modifier le code de la fonction **creation** ainsi que celui de la fonction **main()**.

Code complet avec la fonction **creation** selon le principe First In First Out (LIFO), l'ajout se fait en queue de liste :

```
#include <stdio.h>
#include <stdlib.h>
typedef struct liste{
   int valeur;
   struct liste *suiv;
}Liste;
Liste *tete = NULL;
Liste *creation(Liste *tete,Liste **queue, int x); /* Passage par adresse du pointeur queue de
    liste */
```

```
void affichListe(Liste *tete);
main(){
int x;
char reponse = 'o';
Liste *queue; // Pointeur sur le dernier élément de la liste
do
  {printf("Entrer la valeur : ");
   scanf("%d", &x);
   //Actualisation de la valeur de tete et récupération de la valeur de queue
   tete = creation(tete,&queue,x);
   printf("    Voulez vous continuer o/n ? : ");
   fflush(stdin);/*Vider le buffer en cas de non lecture de la réponse */
   reponse = getchar();
   } while (reponse == 'o');
if(queue!=NULL)queue->suiv = NULL; // Marquer la fin de la liste
printf("\nContenu de la liste :\n");
affichListe(tete);
}
Liste *creation(Liste *tete, Liste **queue,int x){
Liste *p =  (Liste*) malloc(sizeof(Liste));
p->valeur = x;
p->suiv = NULL;
if(tete==NULL) // Si la liste est vide, le nouvel élément devient tête de liste
   tete=p;
   else
   (*queue)->suiv = p; /* Sinon le nouvel élément est chaîné au dernier élément de la liste */
*queue = p; // Actualisation du pointeur queue qui devient le nouvel élément
return tete; // Nouvelle tête si la liste était réellement vide
}
void affichListe(Liste *tete){
 Liste *courant = tete;
 if(courant==NULL)
    printf("Rien a afficher, la liste est vide!\n");
    else
    while(courant != NULL)
     {
      printf("%d  ", courant->valeur);
      courant=courant->suiv;
     }
 printf("\n");}
```

Résultat de l'exécution :

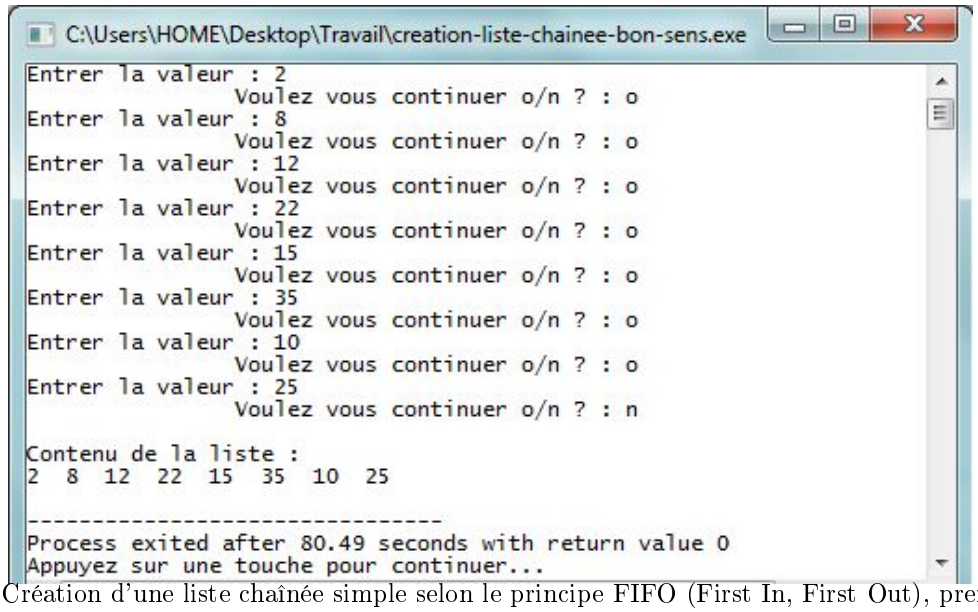


FIGURE 3.9 – Création d'une liste chaînée simple selon le principe FIFO (First In, First Out), premier entré sera premier dans la liste.

Insertion d'une valeur dans une liste : l'insertion d'une valeur dans une liste peut se faire en tête de liste, en queue de liste ou quelque part dans la liste étant donnée la position l ou l'adresse p de l'élément après lequel doit se faire l'insertion.

Insertion en tête de liste :

L'insertion en tête de liste consiste à placer l'élément ajouté au début de la liste qu'elle soit vide ou non. Si elle n'est pas vide, l'élément qui était en tête de liste devient deuxième dans la liste comme illustré dans le Figure 3.10.

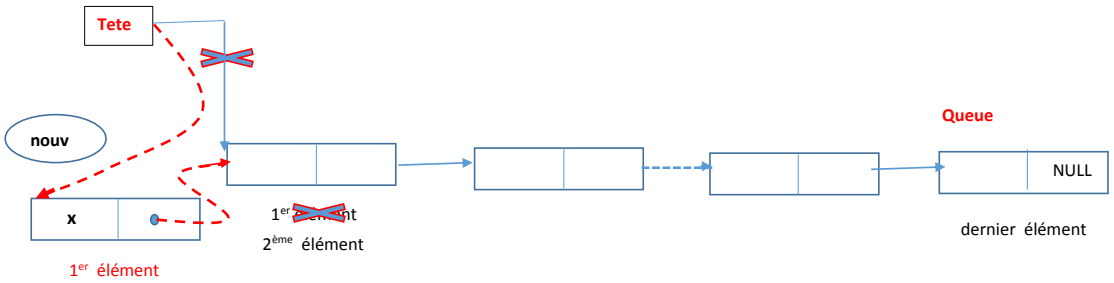


FIGURE 3.10 – Insertion en tête de liste.

L'insertion se fait en appliquant les trois étapes suivantes :

- Créer un nouvel espace mémoire et récupérer son adresse par la fonction **malloc()** ou la fonction **new()** du C++.
- Saisir les informations du nouveau nœud ainsi crée.
- Effectuer le chaînage du nouveau nœud avec le reste de la liste.

Appel dans la fonction main() :

```
tete = insert_tete(tete,x);
```

La fonction insert_tete est comme suit :

```
Liste *insert_tete(Liste *tete, int x)
//Création d'un nouveau noeud
Liste *nouv=(Liste*) malloc(sizeof(Liste));
//Saisir les informations
nouv->valeur = x;
//Chaînage du nouvel élément à la liste
nouv->suiv = tete;//Le premier élément devient deuxième et ainsi de suite
return nouv; // La nouvelle valeur de tete c'est nouv
}
```

Insertion en queue de liste :

L'ajout d'un élément en queue de liste se fait en tête de liste si la liste est vide ou après le dernier élément si la liste contient au moins un élément comme illustré dans la Figure 3.11.

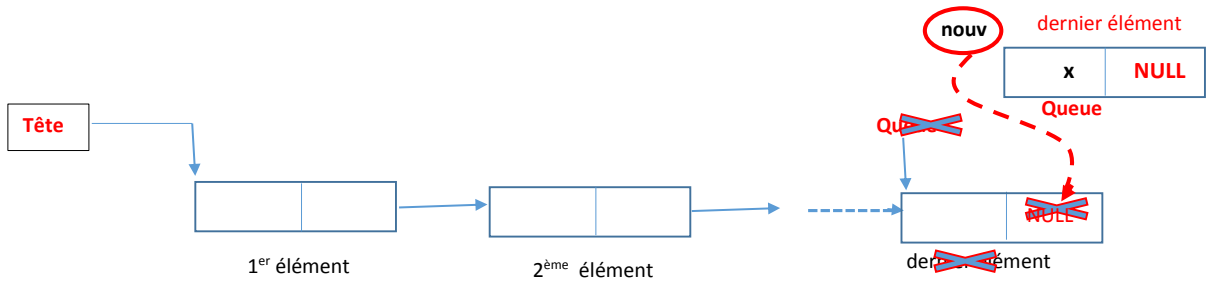


FIGURE 3.11 – Insertion en queue de liste.

Les étapes sont comme suit :

- Créer un nouvel espace mémoire et récupérer son adresse ;
- Saisir les informations du nouveau nœud ainsi crée ;
- Se positionner sur le dernier élément de la liste puis effectuer le chaînage du nouveau nœud avec le reste de la liste.

Appel dans la fonction main() :

```
tete = insert_queue(tete,x);
```

Code de la fonction insert_queue :

```
Liste *insert_queue(Liste *tete, int x)
\\ Déclaration d'un deuxième pointeur sur l'élément courant
Liste *courant;
//Création d'un nouveau noeud
Liste *nouv=(Liste*) malloc(sizeof(Liste));
//Saisir les informations
nouv->valeur = x;
nouv->suiv=NULL;
if(tete==NULL)
   tete = nouv;
    else {
```

```
  // Se positionner sur le dernier élément
  courant = tete:
  while(courant->suiv!=NULL)
   courant = courant->suiv;
  //Chaînage du nouvel élément à la liste
  courant->suiv = nouv; }
return tete; // Nouvelle valeur de tete au cas où la liste était vide
}
```

La fonction peut être donnée de manière récursive comme suit :

```
struct liste *insert_queue(struct liste *tete, int x)
{if(tete == NULL) {
    struct liste *nouv = new liste;
    nouv->valeur = x;
    nouv ->suiv = NULL;
    tete=nouv ;
    }
    else
    tete->suiv = insert_queue(tete->suiv, x);
return tete;
}
```

Insertion à une position l de la liste :

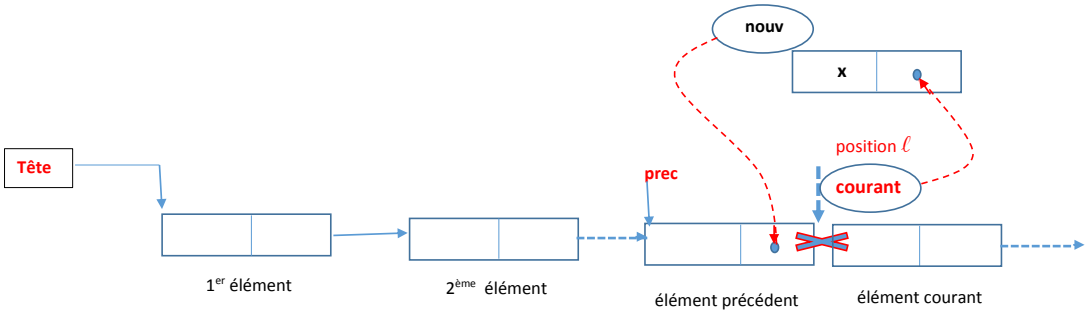


FIGURE 3.12 – Insertion à une position l de la liste, l donnée.

Optons cette fois ci pour une fonction void, la fonction **insert** dont le code est donné en C puis en C++ pour simplifier le code du point de vue appel par adresse ou référence :
En C :
Appel dans la fonction main() :

```
insert(&tete,x,l);
```

```
void insert(Liste **tete, int x, int l) /* Noter que tete est précédée de 2 étoiles l'une
    indiquant que c'est un pointeur et l'autre marquant l'appel par adresse pour récupérer sa
    nouvelle valeur dans le cas d'une éventuelle modification */
{ int i; //Compteur jusqu'à la valeur l
\\ Déclaration de deux pointeurs
```

```
Liste *prec,*courant; /* adresses de l'élément précédent et de l'élément courant (les 2 futurs
    voisins de l'élément nouv) */
//Création d'un nouveau noeud
Liste *nouv=(Liste*) malloc(sizeof(Liste));
//Saisir les informations
nouv->valeur = x;
if(l==1)
 {nouv->suiv=*tete;/*Le pointeur tete apparaît toujours avec une étoile (appel par adresse)*/
   *tete = nouv; // La nouvelle tête de liste
 }
   else
   { // Se positionner à la position l
     i=1;
     courant = *tete;
      while(courant!=NULL && i<l)
        { prec = courant;
         courant = courant->suiv;
         i++;
   }
   if(i==l)
    { nouv->suiv = courant;
     prec->suiv = nouv;
   }
   else
   printf("La liste est trop courte pour insérer a la position %d\n",l);
}
```

En C++ :

```
insert(tete,x,l); // L'opérateur & est omis en C++
```

Le code de la fonction **insert** est comme suit :

```
void insert(Liste *&tete, int x, int l) /* Noter que tete est précédée d'une étoile indiquant
    que c'est un pointeur et & marquant l'appel par adresse */
{ int i; //Compteur jusqu'à la valeur l
\\ Déclaration de deux pointeurs
Liste *prec,*courant; /* adresses de l'élément précédent et de l'élément courant (les 2 futurs
    voisins de l'élément nouv) */
//Création d'un nouveau noeud
Liste *nouv= new Liste;
//Saisir les informations
nouv->valeur = x;
if(l==1)
  { nouv->suiv = tete;
    tete = nouv; // La nouvelle tête de liste
   }
   else
   { // Se positionner à la position l
     i=1;
     courant = tete;
      while(courant!=NULL && i<l)
        { prec = courant;
         courant = courant->suiv;
```

```
        i++;
   }
   if(i==1)
    { nouv->suiv = courant;
      prec->suiv = nouv;
   }
   else
   printf("La liste est trop courte pour insérer a la position %d\n",1);}
```

Insertion après un élément d'adresse p :

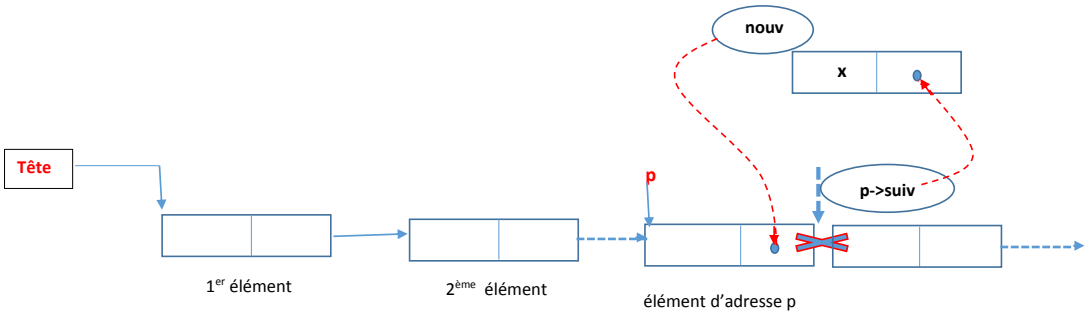


FIGURE 3.13 – Insertion après l'élément d'adresse p, p donnée.

Appel dans la fonction main() :

```
tete=insert(tete,p,x);
```

Le code de la fonction **insert** est comme suit :

```
Liste *insert(Liste *tete, Liste *p, int x){
//Création d'un nouveau noeud
Liste *nouv=(Liste*) malloc(sizeof(Liste));
//Saisir les informations
nouv->valeur = x;
if(p==NULL)
  { nouv->suiv = tete;
     tete = nouv; // La nouvelle tête de liste
   }
   else
    { nouv->suiv = p->suiv;
     p->suiv = nouv;
   }
   return tete;}
```

Á ce stade, il est plus ou moins aisé d'écrire les opérations en C/C++. Jusqu'ici, nous n'avons fait que créer des nœuds et les chaîner entre eux alors qu'en ait-il des autres opérations telles que la recherhce et la suppression dans une liste chaînée ?

Recherche d'une valeur dans une liste : la recherche d'une valeur dans la liste se fait séquentiellement à partir de la tête. Le résultat de la recherche peut être la position (numéro d'ordre dans la liste) du nœud contenant la valeur en question ou bien l'adresse du nœud contenant la valeur. La recherche peut ce faire aussi en spécifiant le nombre d'occurrences d'une valeur dans la liste.

Code de la fonction **recherche** retournant l'adresse de l'élément contenant la valeur val cherchée :
Appel de la fonction :

```
p = recherche(tete,val); // p adresse de l'élément dont la valeur est val
```

```
Liste *recherche(Liste *tete, int val){
Liste *courant = tete;
while(courant != NULL) // Parcours de la liste
  {if(courant->valeur == val)
       return courant;
      courant = courant->suiv;
   }
return NULL; // Liste épuisée et val introuvable
}
```

Suppression d'une valeur dans une liste : la suppression d'une valeur d'une liste consiste à rechercher la valeur et une fois trouvée, supprimer le nœud associé pour libérer la mémoire occupée. La recherche peut se faire par valeur, adresse ou position. Bien entendu, on peut étudier tous les cas de figures (suppression en tête, suppression en queue,...etc) comme pour l'insertion. Considérons le cas d'une suppression connaissant la valeur val de l'élément comme illustré dans la Figure 3.14, dans ce cas la valeur val peut être celle de l'élément se trouvant en tête de liste ou celle du dernier élément de la liste ou bien celle d'un élément se trouvant quelque part dans la liste, comme elle peut ne pas exister du tout. Ainsi le code serait comme suit :

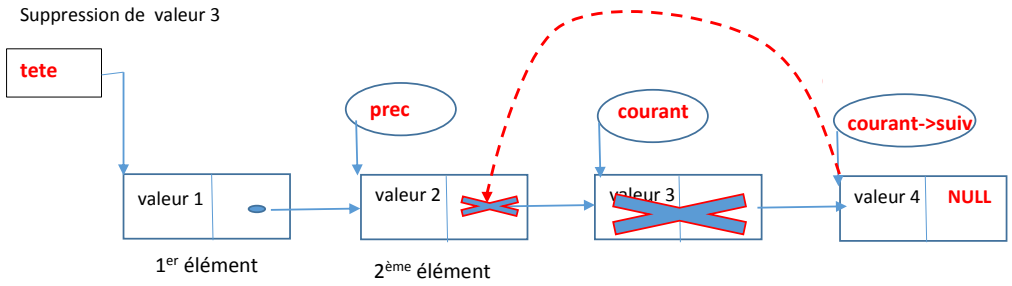


Figure 3.14 – Suppression d'une valeur val, val donnée.

Appel de la fonction :

```
tete = suppression(tete,val);
```

```
Liste *suppression(Liste *tete, int val){
Liste *cour,*prec; /* Le pointeur cour est le pointeur courant et le pointeur prec sert à mé
    moriser l'adresse de l'élément précédent */
if(tete->valeur==val) // La valeur val se trouve en tête de liste
  { prec = tete;
     tete = tete->suiv; // Le deuxième élément devient tête de liste
    free(prec); // Suppression de l'espace alloué
   }
   else
   { prec = tete; // Sinon parcours de la liste à partir du début pour localiser val
     cour = tete->suiv;
      while(cour!=NULL && cour->valeur!=val)
        { prec = cour;
           cour = cour->suiv;
      }
if(cour==NULL)
  printf("La valeur n'existe pas!\n");
   else
   { prec->suiv = cour->suiv; // La valeur val est trouvée à la position indiquée par cour
     free(cour);
   }}
```

La fonction peut être écrite de manière récursive comme suit :

```
truct liste *suppression(struct liste *tete, int x)
{struct liste  *prec;
 if(tete == NULL)
    return NULL;
    else
    if(tete->valeur == x)
    { prec = tete->suiv;
      free(tete);
      tete=prec;
    }
    else
     tete->suiv = suppression(tete->suiv, x);
return tete;
}
```

Fusion de deux listes : la fusion de deux listes consiste à regrouper le contenu de deux listes en une seule, elle peut se faire aléatoirement (coller simplement les listes l'une derrière l'autre) ou bien en respectant l'ordre dans les listes si les listes sont ordonnées. Le code suivant fusionne deux listes ordonnées en une seule liste ordonnée.

Appel dans la fonction main() :

```
tete = fusion(tete1,tete2); // Le pointeur tete appartient à la liste résultat
```

```
Liste *fusion(Liste *tete1, Liste *tete2){
Liste *cour,*cour1,*cour2; // Le pointeur cour parcourt la liste résultat
```

```
cour1=tete1; // Le pointeur cour1 parcourt la première liste dont la tete est tete1
cour2=tete2; // Le pointeur cour2 parcourt la deuxième liste dont la tete est tete2
if(tete2->val>tete1->val) //La valeur du pointeur tete est soit tete1 soit tete2
    {
     tete = tete1;
     cour = tete;
     cour1=tete1->suiv;
    }
    else
    {
    tete = tete2;
    cour = tete;
    cour2=tete2->suiv;
   }
// Ayant obtenu l'élément en tête, construire le reste de la liste
while(cour1 != NULL && cour2 != NULL)
    if(cour2->val>cour1->val)
       {
        cour->suiv = cour1;
        cour=cour1;
        cour1 = cour1->suiv;
       }
       else
       {
       cour->suiv = cour2;
       cour=cour2;
       cour2 = cour2->suiv;
       }
if(cour1==NULL && cour2!=NULL)  // Si la première est épuisée alors chaîner ce qui reste de la
    deuxième
                cour->suiv = cour2;
                else // Si c'est la deuxième qui est épuisée alors chaîner ce qui reste de la
    première
                cour->suiv = cour1;
return tete;
}
```

Code de la fusion aléatoire de deux listes :

```
Liste *fusion(Liste *tete1, Liste *tete2){
while (tete1->suiv!=NULL)
   tete1 = tete1->suiv;
tete1->suiv = tete2;}
```

Suppression d'une liste : la suppression d'une liste consiste à supprimer tous ses nœuds et libérer ainsi l'espace occupé. L'adresse de tête de liste devient alors **NULL**. Le code est comme suit selon qu'on fasse un passage par valeur ou par adresse (référence) du paramètre tete :
Appel dans la fonction main() :

```
tete= supprimer_liste(tete);
```

```
Liste *supprimer_liste(Liste *tete){
Liste *p;
 while(tete!=NULL)
  { p = tete;
     tete = tete->suiv;
   free(p); // libérer l'espace alloué
   }
return NULL;}
```

Passage par adresse ou référence de tete (en C) :
Appel dans la fonction main() :

```
supprimer_liste(&tete);
```

```
void supprimer_liste(Liste **tete){
Liste *p;
 while(*tete!=NULL)
  { p = *tete;
     *tete = (*tete)->suiv;
   free(p); // Libérer l'espace alloué
   }
*tete = NULL;}
```

Passage par adresse ou référence de tete en C++ :
Appel dans la fonction main() :

```
supprimer_liste(tete);
```

```
void supprimer_liste(Liste *&tete){
Liste *p;
 while(tete!=NULL)
  { p = tete;
     tete = tete->suiv;
   free(p); // Libérer l'espace alloué
   }
tete = NULL;}
```

Transformer une liste chaînée en une liste chaînée circulaire simple : une liste chaînée circulaire est une variante d'une liste chaînée dite aussi anneau. Dans cette liste, le dernier élément contient un pointeur sur le premier élément (tête) comme illustré dans la Figure 3.15. Ainsi, la liste peut être parcourue à partir de n'importe quel élément.

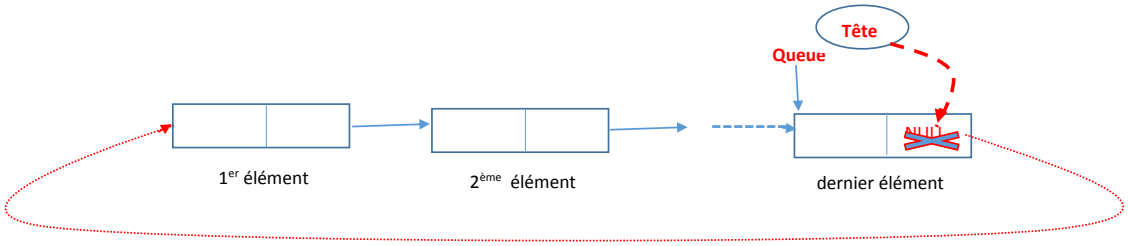


FIGURE 3.15 – Liste circulaire simple.

Code d'une telle transformation :

```
void transformation(Liste *tete){
Liste *courant = tete;
while(courant->suiv!=NULL) // Parcourir la liste et s'arrêter sur le dernier élément
     courant = courant->suiv;
courant->suiv = tete; // Mettre la valeur de tete dans le suivant du dernier élément
}
```

Parcourir et afficher le contenu d'une liste chaînée circulaire simple : le parcours de la liste a lieu à partir de l'élément tête de liste et s'arrête quand on retombe sur lui.

```
void affichListe_circ(Liste *tete){
Liste *p = tete;
if(p==NULL)
    printf("Rien a afficher, la liste est vide!");
    else
   do {printf("%d  ", p->valeur);
       p = p->suiv;
      } while(p!=tete);
printf("\n");}
```

3.6 Listes chaînées bidirectionnelles

Une liste chaînée bidirectionnelle dite aussi doublement chaînée est une liste chaînée où chaque élément contient en plus du pointeur sur l'élément suivant, un pointeur vers l'élément précédent noté **prec** comme illustré dans la Figure 3.16. Ce modèle de liste permet de circuler dans les deux sens. Le parcours des listes peut être aussi facilité par l'utilisation des listes circulaires simplement ou doublement chaînées.

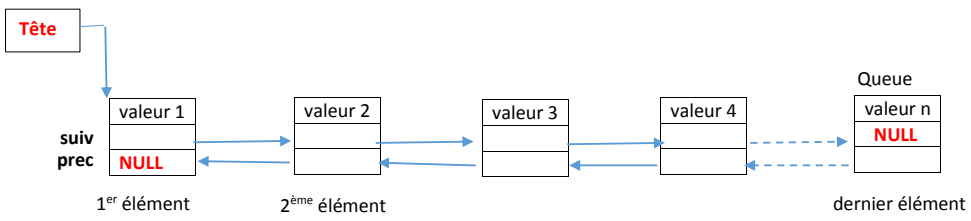


FIGURE 3.16 – Liste chaînée bidirectionnelle.

La déclaration d'une liste doublement chaînée est comme suit :

```
typedef struct liste{
int valeur;
struct liste *prec;
struct liste *suiv;
}Liste;
Liste *tete = NULL; // Initialisation à liste vide
```

3.6.1 Création d'une liste bidirectionnelle

Appel dans la fonction main(), au début le pointeur tete est initialisé à NULL :

```
tete = creation(tete,x) // Où la variable x est la valeur de l'élément créé
```

```
Liste *creation(Liste *tete, int x){
Liste *p =  (Liste*) malloc(sizeof(Liste)); /* Ou bien Liste *p = new Liste tout simplement en
    utilisant l'opérateur new du C++ */
p->val = x;
p->prec = NULL;
p->suiv = tete;
if(tete!=NULL)
   tete->prec = p;
return p; // Nouvelle tête
}
```

3.6.2 Quelques opérations sur une liste bidirectionnelle

Insertion d'un élément après l'élément d'adresse p : cette opération doit mettre à jour les deux pointeurs, celui qui pointe vers l'élément précédent et celui de l'élément suivant.

Appel dans la fonction main() :

```
tete = insertion(tete,p,x); /* Où x est la valeur du nouvel élément */
```

```
Liste* insertion(Liste *tete, Liste *p,int x){
Liste *nouv =  (Liste*) malloc(sizeof(Liste));
nouv->valeur = x;
nouv->prec = p;
if(p==NULL)
  { // Insertion en tête de liste
     nouv->suiv = tete;
      if(tete!=NULL)
         tete->prec = nouv;
      tete = nouv;
   }
   else
   { nouv-suiv p->suiv;
     if(p->suiv!=NULL)
```

```
        p->suiv->prec = nouv;
       p->suiv = nouv;
   }
return tete;
}
```

Suppression d'un élément se trouvant à une position l donnée : cette opération doit également mettre à jour les deux pointeurs et désallouer l'espace occupé par l'élément à supprimer.
Appel dans la fonction main() :

```
tete = suppression(tete,l); /* Où l est la position
de l'élément à supprimer dans la liste */
```

```
void suppression(Liste **tete, int l) /* Passage par adresse de tete au cas d'une éventuelle
    modification */
{Liste *p;
int i; // Compteur jusqu'à l
if(*tete==NULL)
printf("<Rien a supprimer, la liste est vide!\n");
else
{if(l==1)
  { p = *tete;
    *tete = (*tete)->suiv;
   (*tete)->prec = NULL;
   }
   else
 {i = 1;
   p = *tete;
   while(p!=NULL &&i<l)
     { p = p->suiv;
        i++;
      }
 if(p!=NULL)
  { p->suiv->prec = p->prec;
    p->prec->suiv = p->suiv;
  }}
   free(p);
   }}
```

Suppression d'un élément d'adresse p donnée : dans ce cas il n'y a pas de parcours de la liste, la suppression est immédiate.
Appel dans la fonction main() :

```
tete = suppression(tete,p); /* Où p est l'adresse de l'élément à supprimer */
```

```
Liste *suppression(Liste *tete, Liste *p){
if(p->prec!=NULL)
  p->prec->suiv = p->suiv;
 else
```

```
 tete = p->suiv;
if(p->suiv!=NULL)
  p->suiv->prec = p->prec;
free(p);
return tete;
}
```

D'autres traitements peuvent être effectués sur une liste bidirectionnelle tout aussi bien que sur une liste bidirectionnelle circulaire illustrée en Figure 3.17.

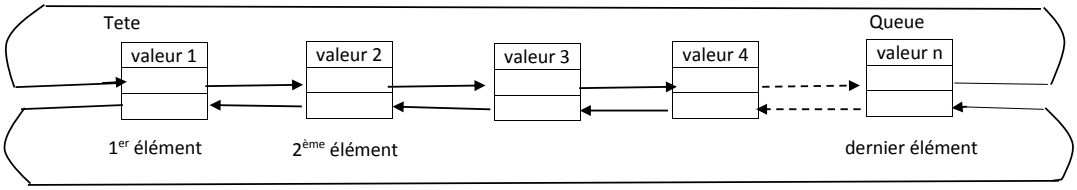


Figure 3.17 – Liste bidirectionnelle circulaire.

Avantages des listes chaînées :

☺ Pas besoin d'indiquer à priori la taille de la liste contrairement à un tableau qui nécessite la taille maximale à l'avance.
☺ Pour l'insertion/suppression d'un élément, il suffit de modifier tout simplement le chaînage contrairement à un tableau qui nécessite le décalage des éléments ce qui est très coûteux en temps.

Inconvénients :

☹ Place supplémentaire occupée pour les pointeurs en plus des données.
☹ L'accès à un élément ne peut se faire qu'après avoir parcouru séquentiellement les éléments qui précèdent, ce qui est pratiquement coûteux en temps si la liste est longue. Alors que dans un tableau l'accès à un élément est direct.

3.7 Structures de données particulières

Les structures de données piles et files sont des types abstraits de données tout comme les listes en général. Elles peuvent être implémentées (représentées en mémoire) en utilisant un tableau (représentation contiguë) ou une liste chaînée. On peut définir un ensemble d'opérations spécifiques en faisant abstraction des détails de l'implémentation. Les opérations pourront par la suite être instancier selon la structure d'implémentation choisie (contiguë ou chaînée).

3.7.1 Les piles

Une pile (stack) est une structure de données fondée sur le principe du dernier arrivé est le premier sorti (ou LIFO pour Last In, First Out). Ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés. L'accès à la structure de données pour une opération d'ajout (insertion) ou de retrait (suppression) d'un élément se fait par une seule extrémité appelée **sommet de la pile** comme illustré dans la Figure 3.18

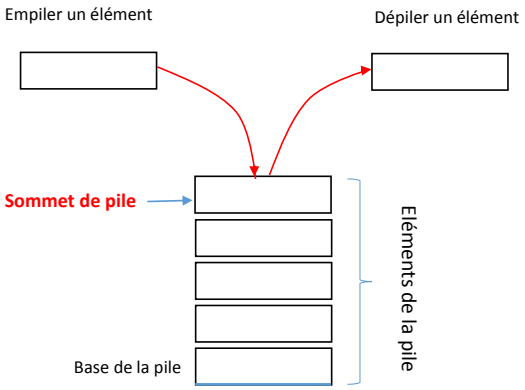


Figure 3.18 – Structure d'une pile.

Les piles sont surtout utilisées pour :

- La gestion des appels des sous-programmes pour sauver le contexte (les paramètres, l'adresse de retour et les variables locales) lors de leur exécution. Ainsi, une fois accomplis, ils sont dépilés. C'est le principe sur lequel repose le déroulement des sous-programmes imbriqués et plus particulièrement récursifs.
- L'analyse syntaxique des expressions parenthésées : à chaque rencontre d'une parenthèse ouvrante, elle sera empilée et à chaque rencontre d'une parenthèse fermante, la dernière parenthèse empilée sera dépilée. Si à la fin la pile est vide, alors l'expression parenthésée est correcte.
- Evaluation des expressions arithmétiques, on distingue les trois notations suivantes :
  - Notation infixée où les opérateurs se placent entre leurs deux opérandes comme $a + b$, $a + b * c$, $(a + b) * c$.
  - Notation préfixée où les opérateurs précèdent la liste de leurs opérandes comme $+ab$, $+a * bc$, $* + abc$.
  - Notation postfixée dite notation polonaise où les opérateurs suivent immédiatement la liste de leurs opérandes comme : $ab+$,$abc * +$, $ab + c*$.
- Parcours en profondeur des arbres (structures de données arborescentes ou hiérarchiques).

Les primitives communément utilisées pour manipuler des piles sont :

pileVide(Pile p) : crée une pile vide p.

sommetPile(Pile p) : renvoie l'élément au sommet de la pile p, p doit être non vide.

empiler(Pile p, Type v) : insère la valeur v au sommet de la pile p.

depiler(Pile p, Type v) : supprime l'élément au sommet de la pile p dont la valeur est récupérée dans la variable v, la pile doit être non vide.

pileVide(Pile p) : teste si la pile p est vide.

3.7.1.1 Implémentation des piles

Les piles peuvent être implémentées soit de manière contiguë (par un tableau), soit par des listes chaînées.

- Implémentation à l'aide d'un tableau : dans cette implémentation les éléments de la pile sont rangés dans un tableau en mémorisant la position du sommet de la pile comme illustré dans la Figure 3.20. La taille maximale du tableau doit être spécifiée à l'avance.

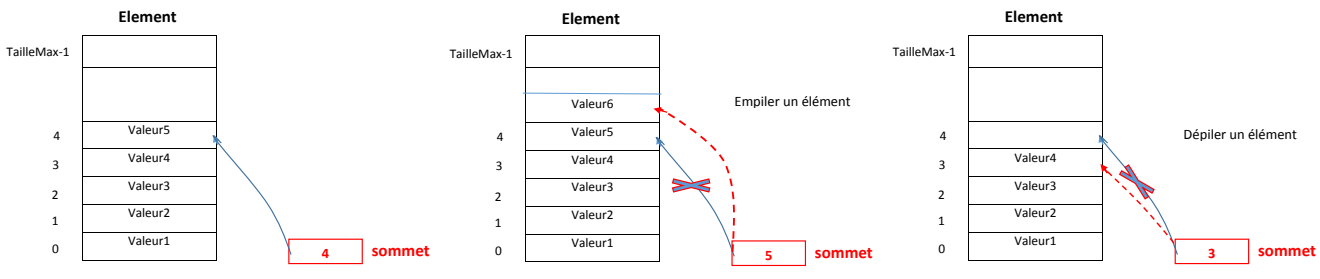


Figure 3.20 – Implémentation d'une pile par un tableau.

Déclaration de la structure pile :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>//Pour utiliser les constantes true et false
#define TailleMax 100 //Taille de la pile figée à l'avance
// Déclaration de la structure Pile
typedef struct{
int Element[TailleMax]; // Tableau de cases où chaque case est un élément de la pile
int ind; // Indice du tableau qui vaut -1 si la pile est vide
}Pile;
```

Implémentation des primitives de manipulation de la pile :

Initialisation de la pile à pile vide : consiste à affecter la valeur -1 à l'indice ind.

```
void initPile(Pile *p) /*Passage par adresse du paramètre p pour récupérer la nouvelle pile p
    après modification*/
{(*p).ind = -1; // Champ indice de la pile initialisé à -1 indiquant que p est vide
}
```

Vérifier si la pile est vide : la pile est vide si l'indice ind vaut -1.

```
bool pileVide(Pile p){
return p.ind==-1;}
```

Vérifier si la pile est pleine : la pile est pleine si la dernière case du tableau (position TailleMax-1) est occupée.

```
bool pilePleine(Pile p){
return (p.ind==TailleMax-1);}
```

Récupération de la valeur du sommet de la pile : la valeur du sommet de la pile est la valeur se trouvant à la position ind du tableau.

```
int sommetPile(Pile p){
return p.Element[p.ind];}
```

Ajouter un élément dans la pile : l'élément est ajouté si la pile n'est pas pleine, l'indice ind est incrémenté indiquant le nouveau sommet de pile.

```
void empiler(Pile *p, int v){
if(!pilePleine(*p))
{ (*p).ind = (*p).ind+1;
   (*p).Element[(*p).ind] = v;
}}
```

Enlever un élément de la pile : la suppression a lieu si la pile n'est pas vide, la fonction appelante va récupérer la valeur de l'élément supprimé et l'indice ind est décrémenté.

```
void depiler(Pile *p, int *v){
if(!pileVide(*p))
{ *v = (*p).Element[(*p).ind];
 (*p).ind = (*p).ind-1;
}}
```

Afficher le contenu de la pile : les éléments sont affichés à partir du sommet de la pile.

```
  void affichPile(Pile p){
int i;
 if(!pileVide(p))
 {
 for(i=p.ind;i>=0;i--)
  printf("\t\t%d\n", p.Element[i]);
 printf("\n");
 }}
```

❀ Implémentation à l'aide d'une liste chaînée : dans cette implémentation le sommet de la pile est une structure à deux champs, un champ valeur où la donnée est stockée et un champ pointeur vers l'élément suivant. La pile peut être caractérisée par sa taille en plus du sommet. La Figure 3.22 montre comment est organisée et gérée la pile ainsi implémentée.

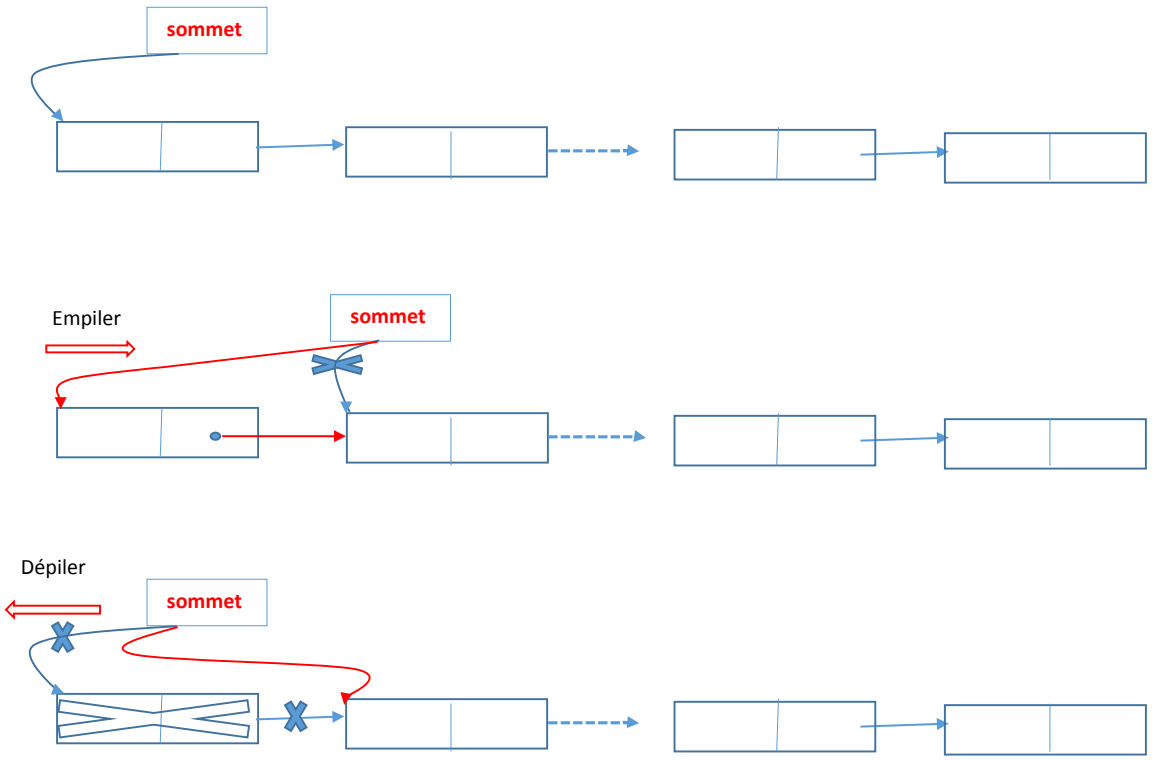


Figure 3.22 – Implémentation d'une pile par une liste chaînée.

Déclaration de la structure pile :

```
#include <stdio.h>
#include <stdlib.h> //Pour utiliser les constantes true et false
// Déclaration de la structure Pile
typedef struct Element{
int valeur; // Valeur de l'élément
struct Element *suiv; //Pointeur sur l'élément suivant
}Element;
// La pile est caractérisée par un sommet et une taille
typedef struct{
Element *sommet;
int taille;
}Pile;
```

Implémentation des primitives de manipulation de la pile :

Initialisation de la pile à pile vide : le pointeur sommet est initialisé à NULL et la taille de la pile à 0.

```
void initPile(Pile *p) /*Passage par adresse du paramètre p pour récupérer la nouvelle pile p
    après modification*/
{ p->sommet = NULL; /*Pointeur sommet de pile initialisé à NULL indiquant pile vide */
  p->taille = 0; // Taille de la pile initialisée à 0
}
```

Vérifier si la pile est vide : il suffit de tester si sa taille est égale à 0 et si l'on veut, le sommet aussi doit pointer sur NULL.

```
bool pileVide(Pile *p){
return (p->taille==0);}
```

Vérifier si la pile est pleine : pas besoin de vérifier si la pile est pleine puisque la taille de la pile n'est pas figée dans le cas d'une implémentation par une liste chaînée.

Récupération de la valeur du sommet de la pile : la valeur du sommet de la pile est retournée à la fonction appelante.

```
int sommetPile(Pile p){
return p.sommet->valeur;}
```

Ajouter un élément dans la pile : consiste à allouer un nouvel espace pour l'élément à ajouter, affecter la valeur v à son champ valeur et mettre à jour le chaînage.

```
void empiler(Pile *p,int v){
  Element *nouv = (Element*) malloc(sizeof(Element));
  if (nouv != NULL){
  nouv->valeur = v;
  nouv->suiv = p->sommet;
  p->sommet = nouv;
  p->taille++;}}
```

Enlever un élément de la pile la suppression d'un élément pourra se faire si la pile n'est pas vide.

```
void depiler(Pile *p,int *v){
Element *sup;
  if(!pileVide(p))
  {
  sup = p->sommet;
  *v = sup->valeur;
  p->sommet = p->sommet->suiv;
  free(sup);
  p->taille--;
  }}
```

Afficher le contenu de la pile : la pile est parcourue à partir du sommet jusqu'au dernier élément dont le suivant pointe sur NULL.

```
   void affichPile(Pile p){
  Element *courant;
  int i; /* Inutile si la condition de la boucle est basée sur le test du sommet de la pile
    par rapport à NULL */
  courant = p.sommet;
  for(i=0;i<p.taille;i++) // Ou bien while(courant!=NULL)
    {
    printf("\t\t%d\n", courant->valeur);
    courant = courant->suiv;
   }}
```

3.7.1.2 Implémentation d'une pile : mise en œuvre

L'implémentation d'une pile est mise en œuvre en utilisant un tableau ; puis en utilisant une liste chaînée.

- Considérons le cas d'une implémentation contiguë : la taille du tableau dans ce cas est fixée à l'avance. Les primitives relatives à cette implémentation définies précédemment sont définies dans un fichier header nommé primitives_Tab.h et leurs prototypes avec la déclaration de la pile dans le fichier nommé prototypes_Tab.h comme suit :
  Fichier prototypes_Tab.h :

```
#define TailleMax 100
typedef struct{
int Element[TailleMax];
int ind;
}Pile;

// Initialisation à pile vide
void initPile (Pile *p);

// Tester si la pile est vide
bool pileVide(Pile p);

// Tester si la pile est pleine
```

```
bool pilePleine(Pile p);

// Empiler un élément
void empiler (Pile *p,int v);

// Dépiler un élément
void depiler (Pile *p,int *v);

// Afficher le contenu de la pile
void affichPile(Pile p);

// Récupérer la valeur du sommet de pile
int sommetPile(Pile p);
```

Fichier primitives_Tab.h :

```
// Initialisation à pile vide
void initPile(Pile *p){
  (*p).ind = -1; }

// Tester si la pile est vide
bool pileVide(Pile p){
 return (p.ind==-1);}

// Tester si la pile est pleine
bool pilePleine(Pile p){
return (p.ind==TailleMax-1);}

// Empiler (ajouter) un élément dans la pile
void empiler(Pile *p,int v){
if(!pilePleine(*p))
{ (*p).ind = (*p).ind+1;
   (*p).Element[(*p).ind] = v;
   }}

// Dépiler (supprimer) un élément de la pile
 void depiler(Pile *p,int *v){
  if(!pileVide(*p))
{ *v = (*p).Element[(*p).ind];
 (*p).ind = (*p).ind-1;
}}

// Afficher le contenu de la pile
void affichPile(Pile p){
 int i;
   if(!pileVide(p))
   for(i=p.ind;i>=0;i--)
    printf("\t\t%d\n", p.Element[i]); }

// Récupérer la valeur du sommet de pile
int sommetPile(Pile p)
{return p.Element[p.ind];}
```

Code de la fonction main() :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "prototypes_Tab.h"
#include "pile_primitives_Tab.h"
int main (){
 Pile p;
 int v, choix;
 char  reponse='o';
 do {printf("\tMenu : implementation contigue (par un tableau)");
     printf("\n\t1 : Creation de la pile");
     printf("\n\t2 : Empiler une valeur");
     printf("\n\t3 : Depiler (sommet de pile)");
     printf("\n\t4 : Quitter");
printf("\n\tFaites votre choix S.V.P : ");
scanf("%d", &choix);
switch (choix)
{case 1 : printf("Creation de la pile\n");
          initPile(&p);
          while(reponse=='o')
         {
         printf ("\nEntrez une  valeur : ");
         scanf ("%d", &v);
         empiler(&p,v);
         printf ("Taille de la pile (%d elements): ",p.ind+1);
         printf("\n__________ Sommet de la PILE __________\n");
         affichPile(p);
         printf("__________ Base de la PILE ____________\n\n");
         fflush(stdin);
         printf("Voulez vous continuer o/n ? :");
         reponse=getchar();
        }
       break;
case 2 : printf("Ajout d'une valeur\n");
         printf ("Entrez la valeur : ");
         scanf ("%d", &v);
         empiler(&p,v);
         printf ("Taille de la pile (%d elements): ",p.ind+1);
         printf("\n__________ Sommet de la PILE __________\n");
         affichPile(p);
         printf("__________ Base de la PILE ____________\n\n");
         break;
case 3 : printf ("\nLe dernier entre [ %d ] sera supprime  (Last In First Out)",sommetPile(p));
         printf ("\nLe dernier entre est supprime\n");
        depiler(&p,&v);  /* Suppression de dernier element entre */
        printf ("Taille de la La pile (%d elements): ",p.ind+1);
        printf("\n__________ Sommet de la PILE __________\n");
        affichPile(p);
        printf("__________ Base de la PILE ____________\n\n");
        break;
default : printf("Fin de traitement, au revoir!\n");
}} while (choix!=4);
}
```

Résultat de l'exécution :

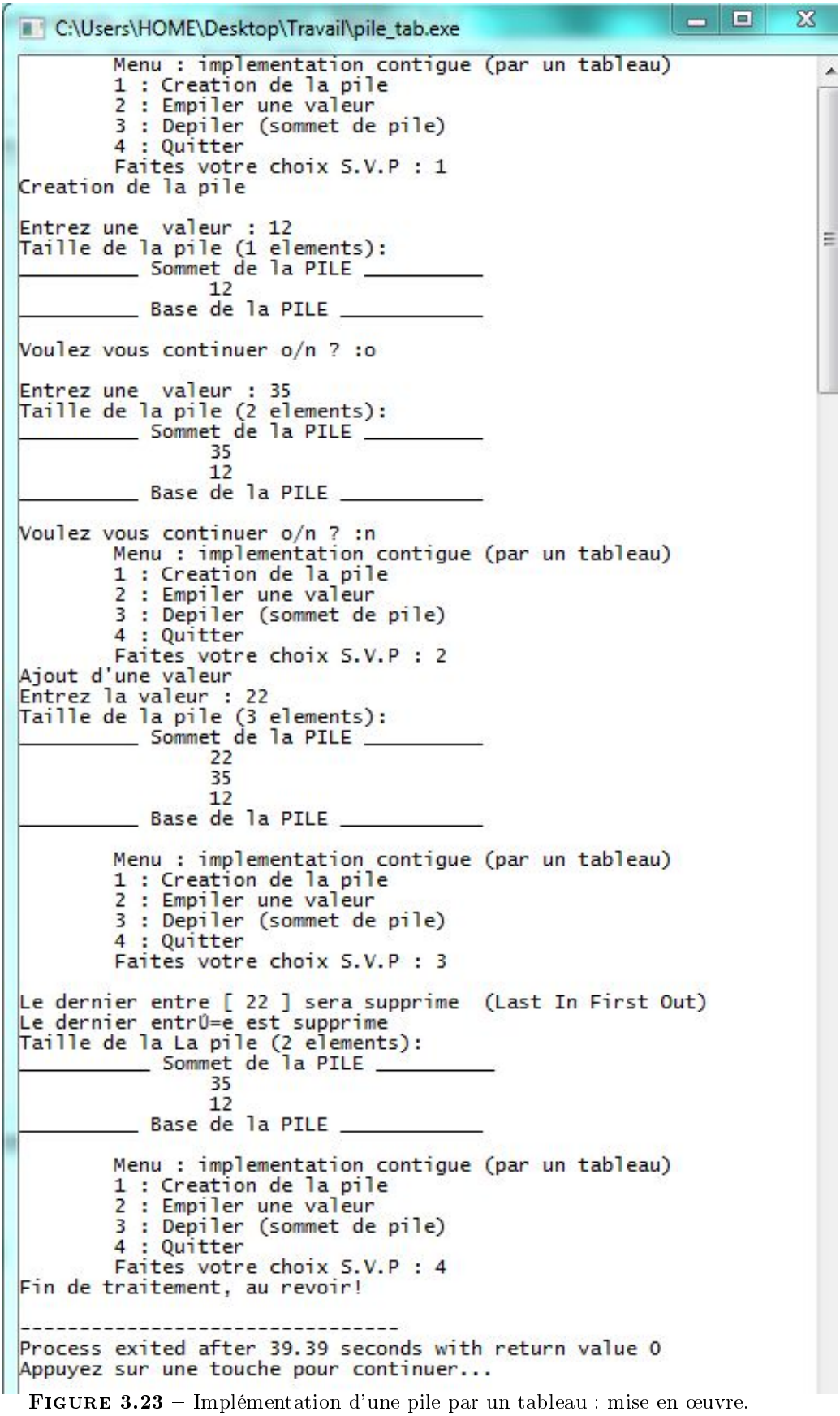


Figure 3.23 – Implémentation d'une pile par un tableau : mise en œuvre.

❀ Considérons la cas d'une implémentation d'une pile par une liste chaînée et utilisons les primitives définies précédemment. Il est plus pratique d'utiliser des fichiers header (d'extension .h) pour y mettre les codes des primitives et leur en-tête puis les inclure dans le code qui réalise la création de la pile.

Fichier header prototypes.h :

```
// Déclaration de la structure Pile
typedef struct Element{
  int valeur;
  struct Element *suiv;
}Element;

typedef struct{
  Element *sommet;
  int taille;
}Pile;

// Initialisation de la pile à pile vide
void initPile (Pile *p);

// Tester si la pile est vide
bool pileVide(Pile *p);

// Empiler (ajouter) un élément dans la pile
void empiler (Pile *p,int v);

// Dépiler (supprimer un élément de la pile
void depiler (Pile *p,int *v);

// Afficher le contenu de la pile
void affichPile(Pile p);

// Récupérer la valeur du sommet de pile
int sommetPile(Pile p);
```

Fichier header pile_primitives.h :

```
// Initialisation de la pille à pile vide
void initPile(Pile *p){
  p->sommet = NULL;
  p->taille = 0;
}
// Tester si la pile est vide
bool pileVide(Pile *p){
 return (p->taille==0);
}
// Empiler (ajouter) un élément dans la pile
void empiler(Pile *p,int v){
  Element *nouv = (Element*) malloc(sizeof(Element));
  if (nouv != NULL)
  {
  nouv->valeur = v;
  nouv->suiv = p->sommet;
  p->sommet = nouv;
  p->taille++;
}}
// Dépiler (supprimer) un élément de la pile
  void depiler(Pile *p,int *v){
  Element *sup;
  if(!pileVide(p))
```

```
  {
  sup = p->sommet;
  *v = sup->valeur;
  p->sommet = p->sommet->suiv;
  free(sup);
  p->taille--;
  }}

// Afficher le contenu de la pile
void affichPile(Pile p){
  Element *courant;
  int i;
  courant = p.sommet;
  for(i=0;i<p.taille;i++){
    printf("\t\t%d\n", courant->valeur);
    courant = courant->suiv;
}}
// Récupérer la valeur du sommet de pile
int sommetPile(Pile p){
return p.sommet->valeur;
}
```

Code de la fonction main() :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "prototypes.h"
#include "pile_primitives.h"
Pile p;
int v;
int main ()
{int choix;
 char  reponse='o';
 do {printf("Menu : implementation par une liste chainee");
      printf("\n\t1 : Creation de la pile");
      printf("\n\t2 : Empiler une valeur");
      printf("\n\t3 : Depiler (sommet de pile)");
      printf("\n\t4 : Quitter");
printf("\n\tFaites votre choix S.V.P : ");
scanf("%d", &choix);
switch (choix)
{
case 1 :  printf("Creation de la pile\n");
          initPile(&p);
          while(reponse=='o')
         {
         printf ("\nEntrez une  valeur : ");
         scanf ("%d", &v);
         empiler(&p,v);
         printf ("Taille de la pile (%d elements): ",p.taille);
         printf("\n__________ Sommet de la PILE __________\n");
         affichPile(p);
         printf("__________ Base de la PILE ____________\n\n");
         fflush(stdin);
```

```
          printf("Voulez vous continuer o/n ? :");
          reponse=getchar();
         }
        break;
case 2 : printf("Ajout d'une valeur\n");
          printf ("Entrez la valeur : ");
          scanf ("%d", &v);
          empiler(&p,v);
          printf ("Taille de la pile (%d elements): ",p.taille);
          printf("\n__________ Sommet de la PILE __________\n");
          affichPile(p);
          printf("__________ Base de la PILE ____________\n\n");
          break;
case 3 : printf ("\nLe dernier entre [ %d ] sera supprime  (Last In First Out)",sommetPile(p));
          printf ("\nLe dernier entre est supprime\n");
         depiler(&p,&v);  /* suppression de dernier element entre */
         printf ("Taille de la La pile (%d elements): ",p.taille);
         printf("\n__________ Sommet de la PILE __________\n");
         affichPile(p);
         printf("__________ Base de la PILE ____________\n\n");
         break;
default : printf("Fin de traitement, au revoir!\n");
}
} while (choix!=4);
}
```

Résultat de l'exécution :

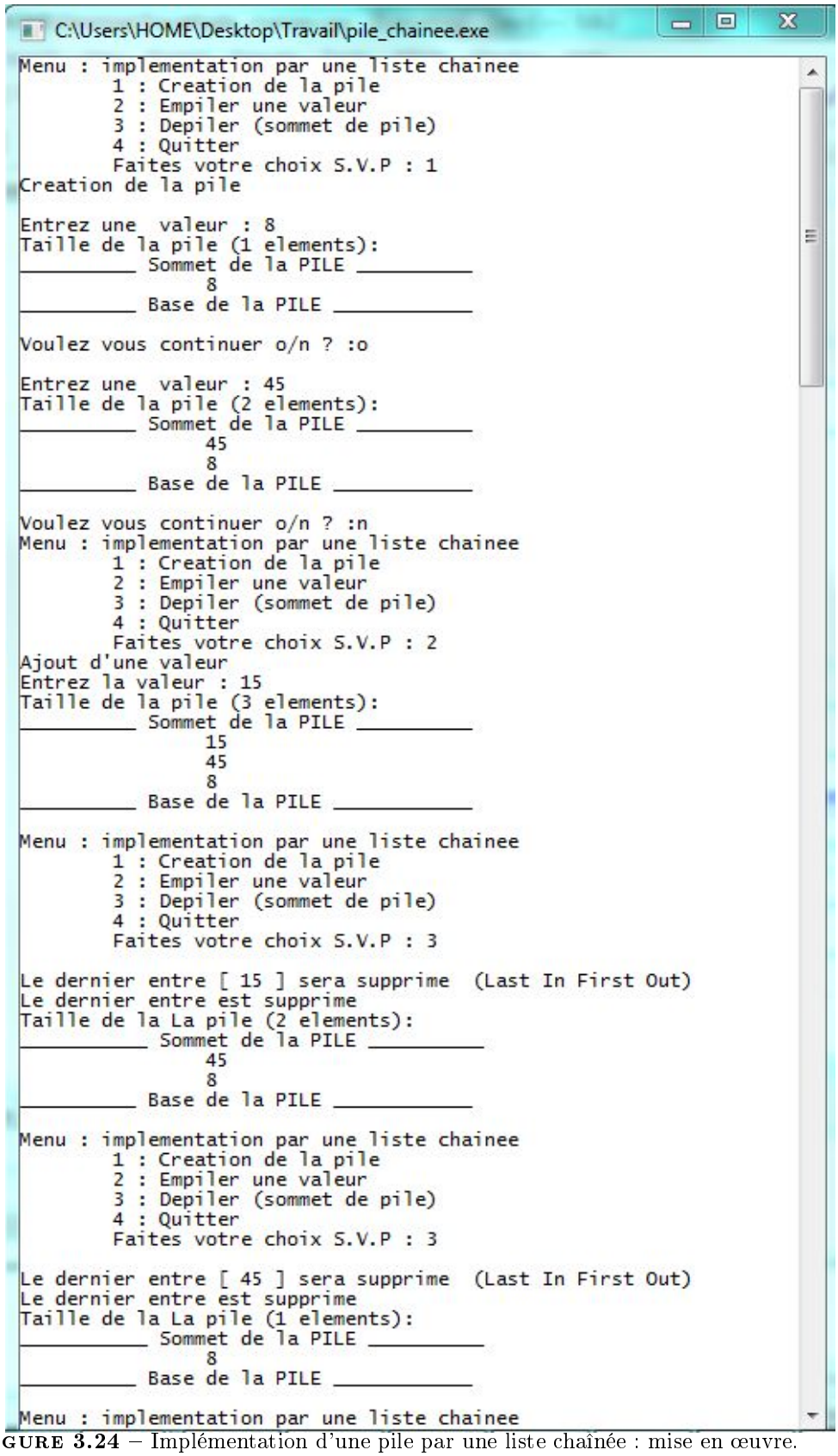


FIGURE 3.24 – Implémentation d'une pile par une liste chaînée : mise en œuvre.

En C++, la structure pile fait partie de la STL (Standard Template Library), il est possible d'y stocker tout type de données (entiers, réels, chaînes de caractères, ...etc). Pour l'utiliser il faut inclure la librairie <stack>, comme suit :

#include <stack>
using namespace std ;
stack<T> p ; /* Où T est le type des éléments de la pile p */

Principales opérations implémentées :

* **push** : enfiler un élément.
* **pop** : défiler un élément.
* **isEmpty** : teste si la file est vide.
* **top** : retourne l'élément au sommet de la pile sans dépilement.
* **size** : retourne la taille de la file (nombre d'éléments).

☞ Exemple de code :

```
#include <stack>
using namespace std;
#include<stdio.h> /* Inutile si on utilise les membres de std cin (lecture) et cout (affichage)
    */
stack<int> p; //pile d'entiers
int main(){
// Afficher la taille de la pile
printf("Le nombre d'elements de la pile est : %d\n",p.size());
// Afficher l'état de la pile (pleine ou vide)
printf("Etat de la pile : ");
if(p.empty())
   printf("La pile est vide\n");
else
 printf("La pile n'est pas vide\n");
// Ajouter des éléments dans la pile
printf("\nEmpiler 2");
p.push(2); // Empiler 2
printf("\nEmpiler 5");
p.push(5); // Empiler 5
printf("\nEmpiler 24");
p.push(24); // Empiler 24
printf("\nValeur du sommet de pile = %d\n",p.top());
// Suppression d'un element
printf("\nDepiler");
p.pop(); // suppression du dernier élément
printf("\nValeur du nouveau sommet = %d\n", p.top());
// Suppression d'un deuxième élément
printf("\nDepiler");
p.pop(); // suppression du dernier élément
printf("\nValeur du nouveau sommet = %d\n", p.top());
}
```

Résultat de l'exécution :

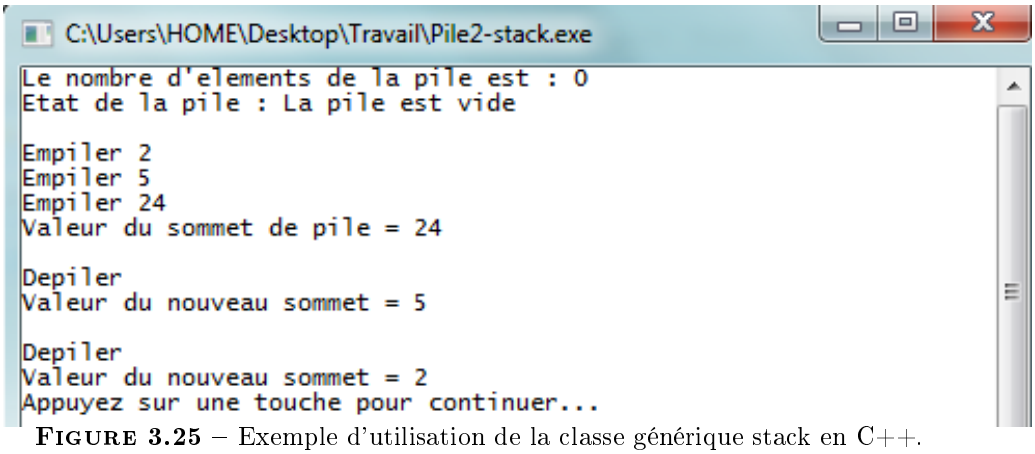


FIGURE 3.25 – Exemple d'utilisation de la classe générique stack en C++.

3.7.1.3 Quelques exemples de traitements sur les piles

Les traitements qu'on peut réaliser sur des piles sont nombreux, proposons quelques exemples :

☞ Exemple1 : suppression des valeurs négatives dans une pile p.

```
-----------------------------
Pile p; //Déclarée globale telle que toute fonction définie dans les exemples peut en disposer
void supValNeg(){
int v;
Pile p1; // Pile de travail créée temporairement
initPile(p1); // Initialiser p1 à pile vide
while(!pilevide(p))
  {depiler(&p,&v);
   if(v>=0)
     empiler(&p1,v);
   }
while(!pileVide(p1))
  {depiler(&p1,&v);
    empiler(&p,v);
   }
}
```

☞ Exemple2 : recherche d'une valeur val dans une pile p de nombres entiers.

a/ Pile p non triée :

```
-----------------------------
Pile p;
bool recherche(int val){
int v;
Pile p1;
initPile(&p1);
while(!pileVide(p) && sommetPile(p)!=val)
  { depiler(&p,&v);
     empiler(&p1,v);
   }
if(!pileVide(p)) // La valeur val est trouvée
{ while(!pileVide(p1))
    {depiler(&p1,&v);
       empiler(&p,v);
     }
```

```
   return true;
}
else
{ while(!pileVide(p1))
    {depiler(&p1,&v);
       empiler(&p,v);
      }
   return false;
}}
```

b/ Pile p triée (le sommet de pile ayant la plus petite valeur) :

```
----------------------------
Pile p;
bool recherche(int val){
int v;
Pile p1;
initPile(&p1);
while(!pileVide(p) && sommetPile(p)<=val)
  { depiler(&p,&v);
     empiler(&p1,v);
   }
if(!pileVide(p) && sommetPile(p)==val)
{ while(!pileVide(p1))
    {depiler(&p1,&v);
       empiler(&p,v);
      }
   return true;
}
else
{ while(!pileVide(p1))
    {depiler(&p1,&v);
       empiler(&p,v);
      }
   return false;
}}
```

☞ Exemple3 : chercher le minimum dans une pile d'entiers puis le supprimer.

```
----------------------------
Pile p;
void supPileMin(){
int v,min;
Pile p1;
initPile(&p1);
if(!pileVide(p))
  { depiler(&p,&min);
     empiler(&p1,min);
   }
while(!pileVide(p))
 { depiler(&p,&v);
    empiler(&p1,v);
   if(v<min)
```

```
     min = v;
}
while(!pileVide(p1))
    {depiler(&p1,&v);
       if(v!=min)
         empiler(&p,v);
      }}
```

3.7.2 Les files

Une file (queue) est une structure de données basée sur le principe du premier entré est premier sorti (ou FIFO pour First In, First Out). Ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés. L'ajout d'un élément se fait à une extrémité appelée queue et le retrait d'un élément se fait à l'autre extrémité appelée tête comme illustré dans la Figure 3.26. Un exemple typique est la file d'attente où le traitement s'effectue selon l'ordre d'arrivée, situation courante de notre quotidien.

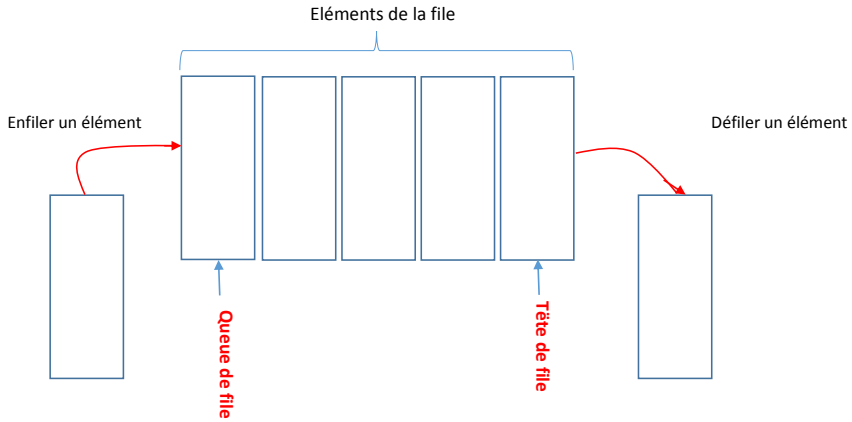


Figure 3.26 – Structure d'une file.

Les files sont surtout utilisées dans :

- La gestion des attentes à un traitement comme des clients devant un guichet, des patients chez un medecin, ...etc.
- Les systèmes d'exploitation des ordinateurs.
- Les tâches d'impression dans un système de traitement par lot.
- Parcours en largeur d'un arbre.

Primitives communément utilisées pour manipuler des files :

initFile(File f) : crée une file vide f.

TeteDeFile(File f) : renvoie l'élément en tête (premier) de la file f, f doit être non vide.

QueueDeFile(File f) : renvoie l'élément en queue (dernier) de la file f, f doit être non vide.

enfiler(File f, Type v) : insère la valeur v à la fin de la file f.

defiler(File f, Type v) : supprime l'élément en tête de la file f dont la valeur est récupérée dans la variable v, la file doit être non vide.

fileVide(File f) : teste si la file f est vide.

3.7.2.1 Implémentation des files

Tout comme la structure pile, la structure file peut être implémentée de façon contiguë ou par une liste chaînée.

- Implémentation à l'aide d'un tableau : cette implémentation nécessite un indice entier qui pointe vers le premier élément dans la file (tete) qui est fixe et égale à 0 et un deuxième indice entier qui pointe vers le dernier élément (queue) qui est variable. Défiler un élément de la file consiste à faire un décalage des éléments du tableau vers l'avant de façon que le premier élément occupe toujours la position 0 du tableau comme illustré dans la Figure 3.27.

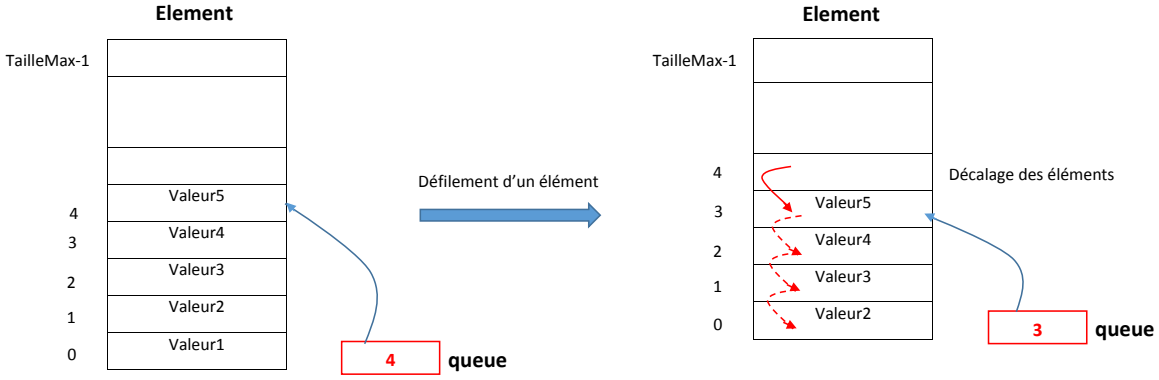


Figure 3.27 – Implémentation d'une file par un tableau avec indice tete fixe et indice queue variable.

Déclaration de la structure file :
Comme l'indice tete vaut toujours 0, il devient inutile dans la déclaration de la structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define TailleMax 100 //Taille de la file figée
// Déclaration de la structure File
typedef struct{
int Element[TailleMax]; // Tableau de cases où chaque case est un élément de la file
int queue; /* Indice de la fin du tableau qui vaut -1 si la file est vide et TailleMax-1 si
    la file est pleine */
}File;
```

Implémentation des primitives de manipulation de la file :

Initialisation de la file à file vide : consiste à positionner l'indice queue à -1 (tableau vide).

```
void initFile(File *f) /* Passage par adresse du paramètre f pour récupérer la nouvelle
    file f après modification */
{ (*f).queue = -1;
}
```

Vérifier si la file est vide : la file est vide si l'indice queue vaut -1.

```
bool fileVide(File f){
return (f.queue==-1);
}
```

Vérifier si la file est pleine : la file est pleine si l'indice queue vaut TailleMax-1.

```
bool filePleine(File f){
return (f.queue==TailleMax-1);
}
```

Récupération de la valeur en tête de file : l'élément en tête de file occupe la position 0.

```
int TeteDeFile(File f){
return f.Element[0];
}
```

Récupération de la valeur en queue de file : l'élément en queue de file peut également être récupéré grâce à l'indice queue.

```
int QueueDeFile(File f){
return f.Element[f.queue];
}
```

Ajouter un élément dans la file : l'élément est ajouté en queue de file. Cette opération est possible si la file n'est pas pleine.

```
void enfiler(File *f, int v){
if(!filePleine(*f))
{ (*f).queue = (*f).queue++;
(*f).Element[(*f).queue] = v;
}
```

Enlever un élément de la file : l'élément est retiré (supprimé) à partie de la tête de la file. Cette opération nécessite le décalage vers l'avant de tous les éléments de la file de sorte que l'élément en tête de file soit toujours à la position fixe 0.

```
void defiler(File *f, int *v){
int i;
if(!fileVide(*f))
{ *v = (*f).Element[0];
for(i=0;i<(*f).queue;i++)
  (*f).Element[i] = (*f).Element[i+1];
(*f).queue = (*f).queue--;
}}
```

Afficher le contenu de la file : la file est parcourue à partir de la position 0 du tableau jusqu'à la position queue.

```
 void affichFile(File f){
int i;
for(i=0;i<=f.queue;i++)
  printf("\t\t%d\n", f.Element[i]);
 }
```

Remarque *: cette implémentation n'est pas efficace puisque le nombre de décalages est d'autant plus important quand la file est longue.*

❀ Implémentation à l'aide d'un tableau avec les deux indices de tete et queue variables dite implémentation par flots : cette implémentation nécessite donc les deux indices tete et queue. Défiler un élément de la file consiste à incrémenter l'indice tete sans avoir à décaler les éléments de la file et enfiler un élément consiste à incrémenter l'indice queue, comme illustré dans la Figure 3.28.

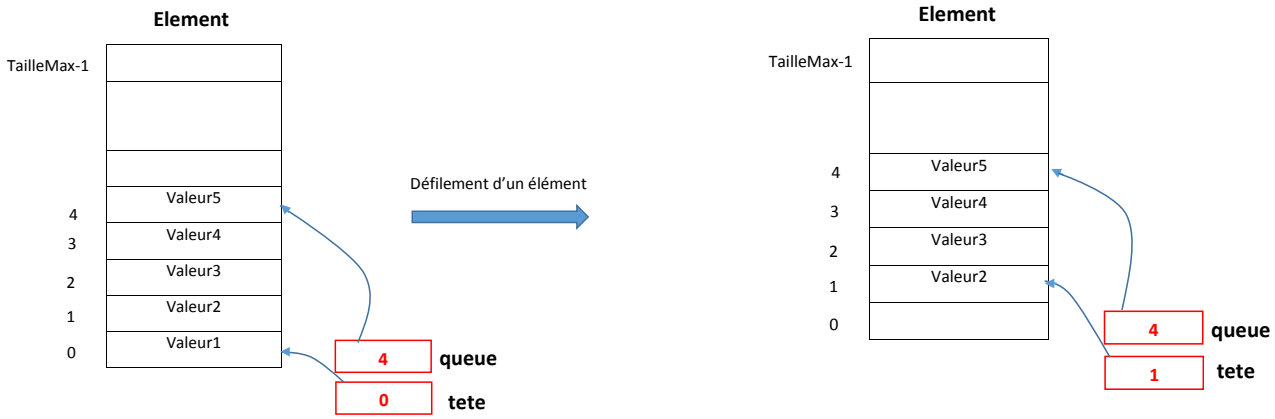


Figure 3.28 – Implémentation d'une file par un tableau avec indices tete et queue variables dite par flots.

Déclaration de la structure file :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define TailleMax 100 // Taille de la file figée
// Déclaration de la structure File
typedef struct{
int Element[TailleMax]; // Tableau de cases où chaque case est un élément de la file
int tete; // vaut 0 si la file est vide
int queue; //  vaut -1 si la file est vide et TailleMax-1 si la file est pleine
}File;
```

Implémentation des primitives de manipulation de la file :

Initialisation de la file à file vide : consiste à positionner l'indice queue à -1 (tableau vide) et l'indice tete à 0.

```
void initFile(File *f) /* Passage par adresse du paramètre f pour récupérer la nouvelle
    file f après modification */
{ (*f).queue = -1;
  (*f).tete = 0;
  }
```

Vérifier si la file est vide : la file est vide si l'indice queue est inférieur à l'indice tete.

```
bool fileVide(File f){
return (f.queue<f.tete);
}
```

Vérifier si la file est pleine : la file est pleine si l'indice queue vaut TailleMax-1.

```
bool filePleine(File f){
return (f.queue==TailleMax-1);
}
```

Récupération de la valeur en tête de file : l'élément en tête de file occupe la position tete.

```
int TeteDeFile(File f){
return f.Element[f.tete];
}
```

Récupération de la valeur en queue de file : l'élément en queue de file peut également être récupéré grâce à l'indice queue.

```
int QueueDeFile(File f){
return f.Element[f.queue];
}
```

Ajouter un élément dans la file : l'élément est ajouté en queue de file. Cette opération est possible si la file n'est pas pleine.

```
void enfiler(File *f, int v){
if(!filePleine(*f))
{ (*f).queue = (*f).queue++;
(*f).Element[(*f).queue] = v;
}
```

Enlever un élément de la file : l'élément est retiré (supprimé) à partie de la tête de la file. Cette fois ci, il n'y a pas de décalages des éléments de la file mais juste une incrémentation de l'indice tete, ainsi les élément occupant des positions inférieurs à tete sont ignorés (inexploitables), ce qui engendre de l'espace perdu dans le tableau.

```
void defiler(File *f, int *v){
if(!fileVide(*f))
{ *v = (*f).Element[(*f).tete];
 (*f).tete = (*f).tete++;
}
}
```

Afficher le contenu de la file : la file est parcourue à partir de la position tete du tableau jusqu'à la position queue. Le nombre d'éléments est toujours égal à $queue - tete + 1$.

```
 void affichFile(File f){
int i;
for(i=f.tete;i<=f.queue;i++)
  printf("\t\t%d\n", f.Element[i]);
 }
```

Remarque *: cette implémentation ne nécessite pas de décalages mais lors du défilement l'espace occupé est perdu. Pour remédier aux inconvénients des deux implémentations (décalages et espace perdu), on a recours à un tableau circulaire pour réutiliser l'espace perdu par les défilements.*

❀ Implémentation à l'aide d'un tableau circulaire : dans cette implémentation la tête et la queue de la file sont toutes les deux variables. L'idée est que lorsque l'indice queue aura atteint la taille maximale du tableau (TailleMax) et qu'il y ait des cases libres du côté de l'extrémité tete alors on recommence à insérer les éléments à partir du début du tableau. Cela signifie, que la première case du tableau (case d'indice 0) est le prolongement de la dernière case (case d'indice TailleMax-1). Si on considère un indice i du tableau, une fois arrivé à la valeur $TailleMax - 1$, il repasse à la valeur 0 soit à $(i + 1)$ $modulo$ $TailleMax$ (où modulo est l'opérateur donnant le reste d'une division entière). Ainsi, les indices queue et tete sont incrémentés modulo TailleMax, le premier est incrémenté à chaque ajout et le deuxième à chaque suppression. La file est vide si l'indice tete coïncide avec l'indice queue (tete=queue) et pleine si $(queue + 1)$ modulo $TailleMax = tete$, le mécanisme est illustré dans la Figure 3.30.

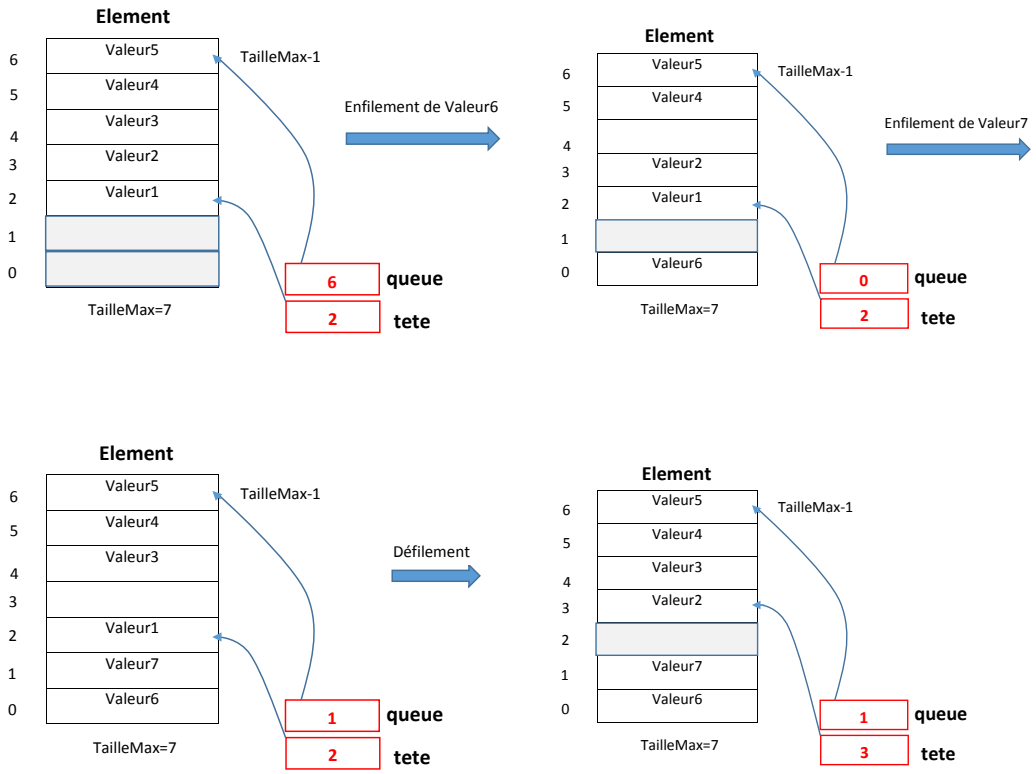


Figure 3.30 – Implémentation d'une file par un tableau circulaire.

Implémentation des primitives de manipulation de la file :

Initialisation de la file à file vide : les deux indices tete et queue sont égaux et valent 0.

```
void initFile(File *f) /* Passage par adresse du paramètre f pour récupérer la nouvelle
    file f après modification */
{(*f).tete = TailleMax-1;
(*f).queue = TailleMax-1;
}
```

Vérifier si la file est vide : la file est vide si les deux indices sont égaux.

```
bool fileVide(File f){
return (f.tete==f.queue);}
```

Vérifier si la file est pleine : la file est pleine si l'indice tete vaut $(queue + 1)$ $modulo$ $TailleMax$.

```
bool filePleine(File f){
return (f.queue+1)%TailleMax == f.tete;}
```

Récupération de la valeur en tête de file : la valeur est récupérée à la position d'indice $(tete + 1)$ $modulo$ $TailleMax$ comme suit :

```
int TeteDeFile(File f){
return f.Element[(f.tete+1)%TailleMax]; }
```

Récupération de la valeur en queue de file : la valeur est récupérée à la position d'indice queue comme suit :

```
int QueueDeFile(File f){
return f.Element[f.queue];
}
```

Ajouter un élément dans la file : l'élément est ajouté en queue de file et l'indice queue est incrémenté modulo TailleMax.

```
void enfiler(File *f, int v){
if(!filePleine(*f))
{ if((*f).queue == TailleMax-1)
      (*f).queue = 0; // Reprend à partir de la position 0
     else
     (*f).queue = (*f).queue+1;
 (*f).Element[(*f).queue] = v;
}}
```

Enlever un élément de la file : l'élément est retiré (supprimé) à partie de la tête et l'indice tete est incrémenté modulo TailleMax.

```
void defiler(File *f, int *v){
if(!fileVide(*f))
```

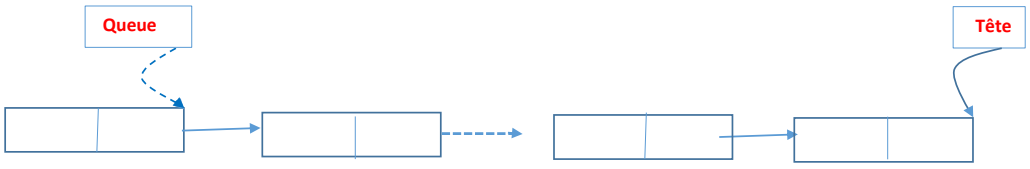
```
{  if((*f).tete==TailleMax-1)
   (*f).tete = 0; // Reprend à partir de la position 0
   else
(*f).tete = (*f).tete+1;
*v = (*f).Element[(*f).tete];
}}
```

Afficher le contenu de la file : pour parcourir le contenu de la file, on parcourt les indices de tete à $(tete + 1)$ *modulo* $TailleMax$ à queue.

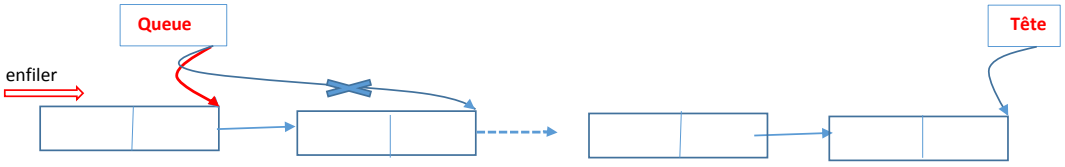
```
void affichFile(File f){
int i;
for(i=f.tete%TailleMax+1;i<=f.queue;i++)
  printf("\t\t%d\n", f.Element[i]);
 }
```

Remarque *: la difficulté de cette implémentation est qu'il n'est pas possible de distinguer entre file vide et file pleine. Ce qui peut être résolu soit en utilisant une variable supplémentaire donnant le nombre d'éléments dans la file, soit en considérant qu'une file qui contient* $TailleMax - 1$ *éléments est déjà pleine.*

- Implémentation à l'aide d'une liste chaînée : tout comme la pile, une file peut être implémentée par une liste chaînée mais doit disposer de deux pointeurs : un pointeur vers la tête de la file et un autre vers la queue de la file. Les insertions se font en queue de file et les suppression en tête de file comme illustré dans la Figure 3.32.



(a) File implémentée par une liste chaînée.



(b) Enfilement dans une file implémentée par une liste chaînée.

(c) Défilement dans une file implémentée par une liste chaînée.

FIGURE 3.32 – Implémentation d'une file par une liste chaînée.

Déclaration de la structure file :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
// Déclaration de la structure file
typedef struct Element{
int valeur; // Valeur de l'élément
struct Element *suiv; // Pointeur sur l'élément suivant
}Element;
// La file est caractérisée par deux extrémités et donc deux pointeurs
typedef struct{
Element *teteFile; // Tête de la file
Element *queueFile; // Queue de la file
}File;
```

Implémentation des primitives de manipulation de la file :

Initialisation de la file à file vide : les deux pointeurs de la file pointent sur NULL.

```
void initFile(File *f)
{
  f->teteFile = NULL;
  f->queueFile = NULL;
}
```

Tester si la file est vide : le file est vide si les deux pointeurs pointent sur NULL.

```
bool fileVide(File f)
{
 return (f.teteFile==NULL && f.queueFile==NULL);
}
```

Enfiler (ajouter) un élément dans la file : un nouvel espace est alloué à l'élément qui est ajouté en queue de file qui devient la nouvelle queue de file et éventuellement la nouvelle tete de la file si la file est vide.

```
void enfiler(File *f, int v)
{
Element *nouv = (Element*) malloc(sizeof(Element));
nouv->valeur = v;
nouv->suiv = NULL;
if(fileVide(*f))
  f->teteFile = nouv;
   else
   f->queueFile->suiv = nouv;
f->queueFile = nouv;
}
```

Défiler (retirer) un élément de la file : l'élément est supprimé à partir de la tête de la file, le chaînage est mis à jour et l'espace libéré.

```
void defiler(File *f,int *v)
{
Element *p;
if(!fileVide(*f))
 { p = f->teteFile;
*v = teteDeFile(*f);
f->teteFile = f->teteFile->suiv;
if(f->teteFile==NULL)
  f->queueFile = NULL;
free(p);
}}
```

Récupérer la valeur de l'extrémité tête de file : la valeur est accessible si la liste n'est pas vide.

```
int teteDeFile(File f)
{
if(!fileVide(f))
return (f.teteFile)->valeur;
}
```

Récupérer la valeur de l'extrémité queue de file la valeur est accessible si la file n'est pas vide.

```
int queueDeFile(File f){
if(!fileVide(f))
return (f.queueFile)->valeur;
}
```

Afficher le contenu de la file : la file est parcourue du premier élément (tête de file) jusqu'au dernier (queue de file) dont le suivant pointe sur NULL.

```
   void affichFile(File f)
   {
  Element *courant;
  courant = f.teteFile;
  while(courant!=NULL)
    {
    printf("\t\t%d\n", courant->valeur);
    courant = courant->suiv;
 }}
```

3.7.2.2 Implémentation d'une file : mise en œuvre

L'implémentation d'une file est mise en œuvre en utilisant un tableau circulaire puis en utilisant une liste chaînée.

- Considérons le cas d'une implémentation contiguë (tableau circulaire) : la taille du tableau dans ce cas est fixée à l'avance. Les primitives relatives à cette implémentation définies précédemment sont définies dans un fichier header nommé file_primitives_Tab.h et leurs prototypes avec la déclaration de la file dans le fichier nommé file_prototypes_Tab.h comme suit :

  Fichier file_prototypes_Tab.h :

```
#define TailleMax 10 // Taille de la file figée
// Déclaration de la structure File
typedef struct{
int Element[TailleMax]; /* Tableau de cases où chaque case est un élément de la file */
int tete;
int queue;
}File;
void initFile(File *f);
bool fileVide(File f);
bool filePleine(File f);
int TeteDeFile(File f);
int QueueDeFile(File f);
void enfiler(File *f,int v);
void defiler(File *f,int *v);
void affichFile(File f);
```

  Fichier file_primitives_Tab.h :

```
void initFile (File *f)
{(*f).tete = TailleMax-1;
(*f).queue = TailleMax-1;
}

bool fileVide(File f){
return (f.queue==f.tete);}

bool filePleine(File f){
return (f.queue+1) % TailleMax == f.tete;}

int TeteDeFile(File f){
return f.Element[(f.tete+1)%TailleMax]; }

int QueueDeFile(File f){
return f.Element[f.queue]; }

void enfiler(File *f, int v){
if(!filePleine(*f))
{ if((*f).queue == TailleMax-1)
      (*f).queue = 0;
     else
     (*f).queue = (*f).queue+1;
 (*f).Element[(*f).queue] = v;
}}
```

```
void defiler(File *f, int *v){
if(!fileVide(*f))
{  if((*f).tete==TailleMax-1)
   (*f).tete = 0;
   else
(*f).tete = (*f).tete+1;
*v = (*f).Element[(*f).tete];
}}

void affichFile(File f){
  int i;
  for(i=(f.tete+1)%TailleMax;i<=f.queue;i++)
    printf("\t\t%d\n", f.Element[i]);
   }
```

Fonction main() réalisant la création :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "file_prototypes_Tab.h"
#include "file_primitives_Tab.h"
File f;
int v;
int main (){
int choix;
 char  reponse='o';
 do {printf("Menu : implementation par un tableau circulaire");
      printf("\n\t1 : Creation de la file");
      printf("\n\t2 : Enfiler une valeur");
      printf("\n\t3 : Defiler une valeur");
      printf("\n\t4 : Quitter");
printf("\n\tFaites votre choix S.V.P : ");
scanf("%d", &choix);
switch (choix)
{
case 1 :  printf("Creation de la file\n");
           initFile(&f);
           while(reponse=='o')
          {
          printf ("\nEntrez une  valeur : ");
          scanf ("%d", &v);
          enfiler(&f,v);
          printf ("Valeur en tete de file (%d): ",TeteDeFile(f));
          printf ("Valeur en queue de file (%d): ",QueueDeFile(f));

          printf("\n---------- Tete de la FILE ----------\n");
          affichFile(f);
          printf("---------- Queue de la FILE ----------\n\n");
          fflush(stdin);
          printf("Voulez vous continuer o/n ? :");
          reponse=getchar();
         }
       break;
case 2 : printf("Ajout d'une valeur\n");
```

```
          printf ("Entrez la valeur : ");
          scanf ("%d", &v);
          enfiler(&f,v);
          printf ("Valeur en tete de file (%d): ",TeteDeFile(f));
          printf ("Valeur en queue de file (%d): ",QueueDeFile(f));
          printf("\n__________ Tete de la FILE __________\n");
          affichFile(f);
          printf("__________ Queue de la FILE __________\n\n");
          break;
case 3 : printf ("\nLe premier entre [ %d ] sera supprime  (First In First Out)",TeteDeFile(f
    ));
          printf ("\nLe premier entre est supprime\n");
         defiler(&f,&v);
         printf ("Valeur en tete de file (%d): ",TeteDeFile(f));
          printf ("Valeur en queue de file (%d): ",QueueDeFile(f));
         printf("\n__________ Tete de la FILE __________\n");
         affichFile(f);
         printf("__________ Queue de la FILE __________\n\n");
         break;
default : printf("Fin de traitement, au revoir!\n");
}
} while (choix!=4);}
```

Résultat de l'exécution :

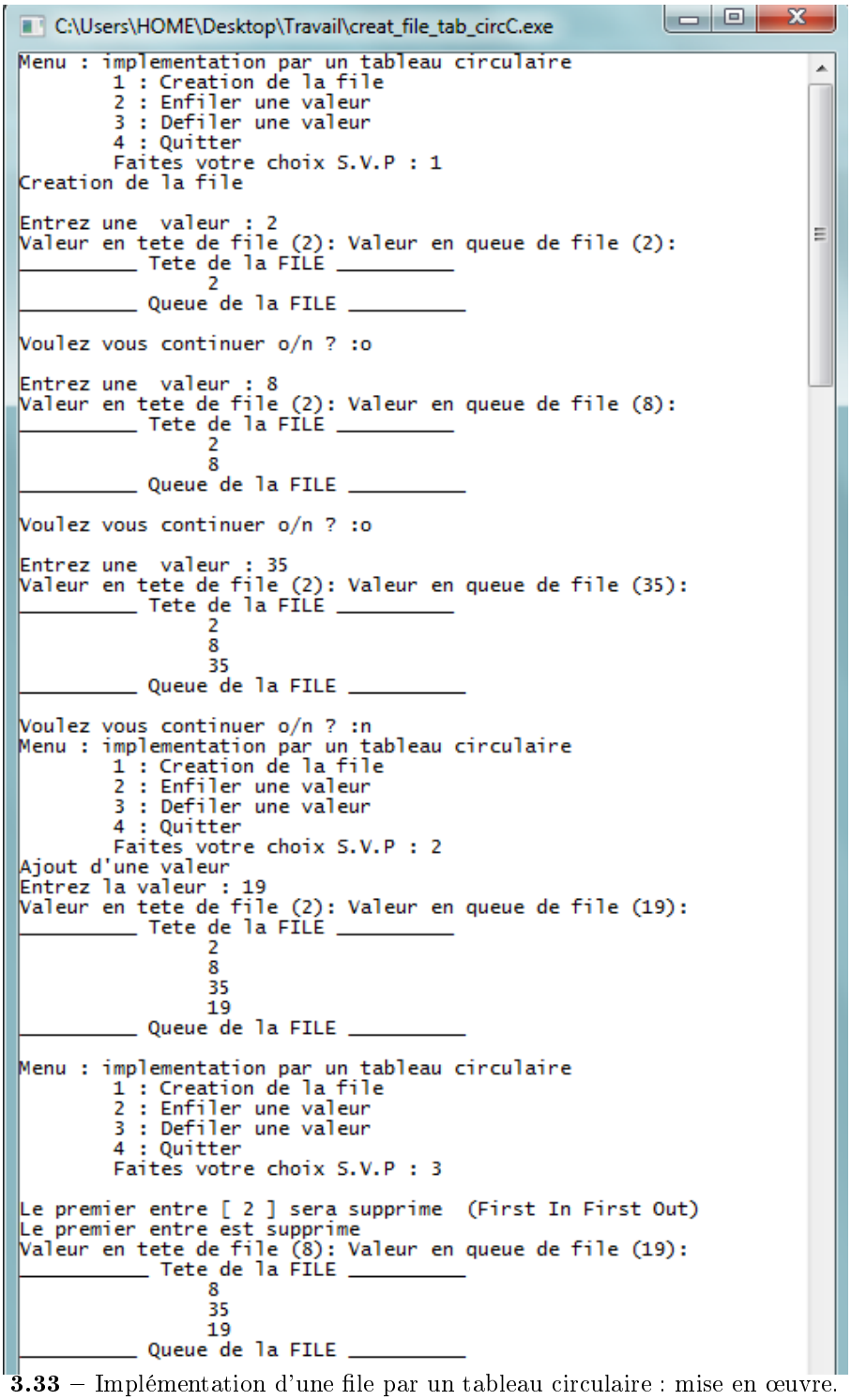


FIGURE 3.33 – Implémentation d'une file par un tableau circulaire : mise en œuvre.

❀ Considérons la cas d'une implémentation d'une file par une liste chaînée et utilisons les primitives définies précédemment.

Fichier header file_prototypes.h :

```
// Déclaration de la structure file
typedef struct Element{
int valeur; // Valeur de l'élément
struct Element *suiv; // Pointeur sur l'élément suivant
}Element;
// La file est caractérisée par deux extrémités et donc deux pointeurs
typedef struct{
Element *teteFile; // Tête de la file
Element *queueFile; // Queue de la file
```

```
}File;
void initFile(File *f);
bool fileVide(File f);
void enfiler(File *f, int v);
void defiler(File *f,int *v);
int teteDeFile(File f);
int queueDeFile(File f);
void affichFile(File f);
```

Fichier header file_primitives.h :

```
void initFile(File *f){
  f->teteFile = NULL;
  f->queueFile = NULL;
}

bool fileVide(File f){
 return (f.teteFile==NULL && f.queueFile==NULL);
}

void enfiler(File *f, int v){
Element *nouv = (Element*) malloc(sizeof(Element));
nouv->valeur = v;
nouv->suiv = NULL;
if(fileVide(*f))
  f->teteFile = nouv;
   else
   f->queueFile->suiv = nouv;
f->queueFile = nouv;
}

void defiler(File *f,int *v){
Element *p;
if(!fileVide(*f))
 { p = f->teteFile;
*v = teteDeFile(*f);
f->teteFile = f->teteFile->suiv;
if(f->teteFile==NULL)
  f->queueFile = NULL;
free(p);
}}

int teteDeFile(File f){
if(!fileVide(f))
return (f.teteFile)->valeur;
}

int queueDeFile(File f){
if(!fileVide(f))
return (f.queueFile)->valeur;
}

void affichFile(File f){
  Element *courant;
  courant = f.teteFile;
```

```
  while(courant!=NULL)
    {
    printf("\t\t%d\n", courant->valeur);
    courant = courant->suiv;
   }}
```

Fonction main() réalisant la création :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "file_prototypes.h"
#include "file_primitives.h"
File f;
int v;
int main ()
{int choix;
 char  reponse='o';
 do {printf("Menu : implementation par une liste chainee");
      printf("\n\t1 : Creation de la file");
      printf("\n\t2 : Enfiler une valeur");
      printf("\n\t3 : Defiler une valeur");
      printf("\n\t4 : Quitter");
printf("\n\tFaites votre choix S.V.P : ");
scanf("%d", &choix);
switch (choix)
{
case 1 :  printf("Creation de la file\n");
           initFile(&f);
           while(reponse=='o')
          {
          printf ("\nEntrez une  valeur : ");
          scanf ("%d", &v);
          enfiler(&f,v);
          printf ("Valeur en tete de file (%d): ",teteDeFile(f));
          printf ("Valeur en queue de file (%d): ",queueDeFile(f));

          printf("\n__________ Tete de la FILE __________\n");
          affichFile(f);
          printf("__________ Queue de la FILE __________\n\n");
          fflush(stdin);
          printf("Voulez vous continuer o/n ? :");
          reponse=getchar();
         }
        break;
case 2 : printf("Ajout d'une valeur\n");
          printf ("Entrez la valeur : ");
          scanf ("%d", &v);
          enfiler(&f,v);
          printf ("Valeur en tete de file (%d): ",teteDeFile(f));
          printf ("Valeur en queue de file (%d): ",queueDeFile(f));
          printf("\n__________ Tete de la FILE __________\n");
          affichFile(f);
          printf("__________ Queue de la FILE __________\n\n");
          break;
```

```
case 3 : printf ("\nLe premier entre [ %d ] sera supprime  (First In First Out)",teteDeFile(f
    ));
         printf ("\nLe premier entre est supprime\n");
        defiler(&f,&v);
        printf ("Valeur en tete de file (%d): ",teteDeFile(f));
         printf ("Valeur en queue de file (%d): ",queueDeFile(f));
        printf("\n__________ Tete de la FILE __________\n");
        affichFile(f);
        printf("__________ Queue de la FILE __________\n\n");
        break;
default : printf("Fin de traitement, au revoir!\n");
}
} while (choix!=4);
}
```

Résultat de l'exécution :

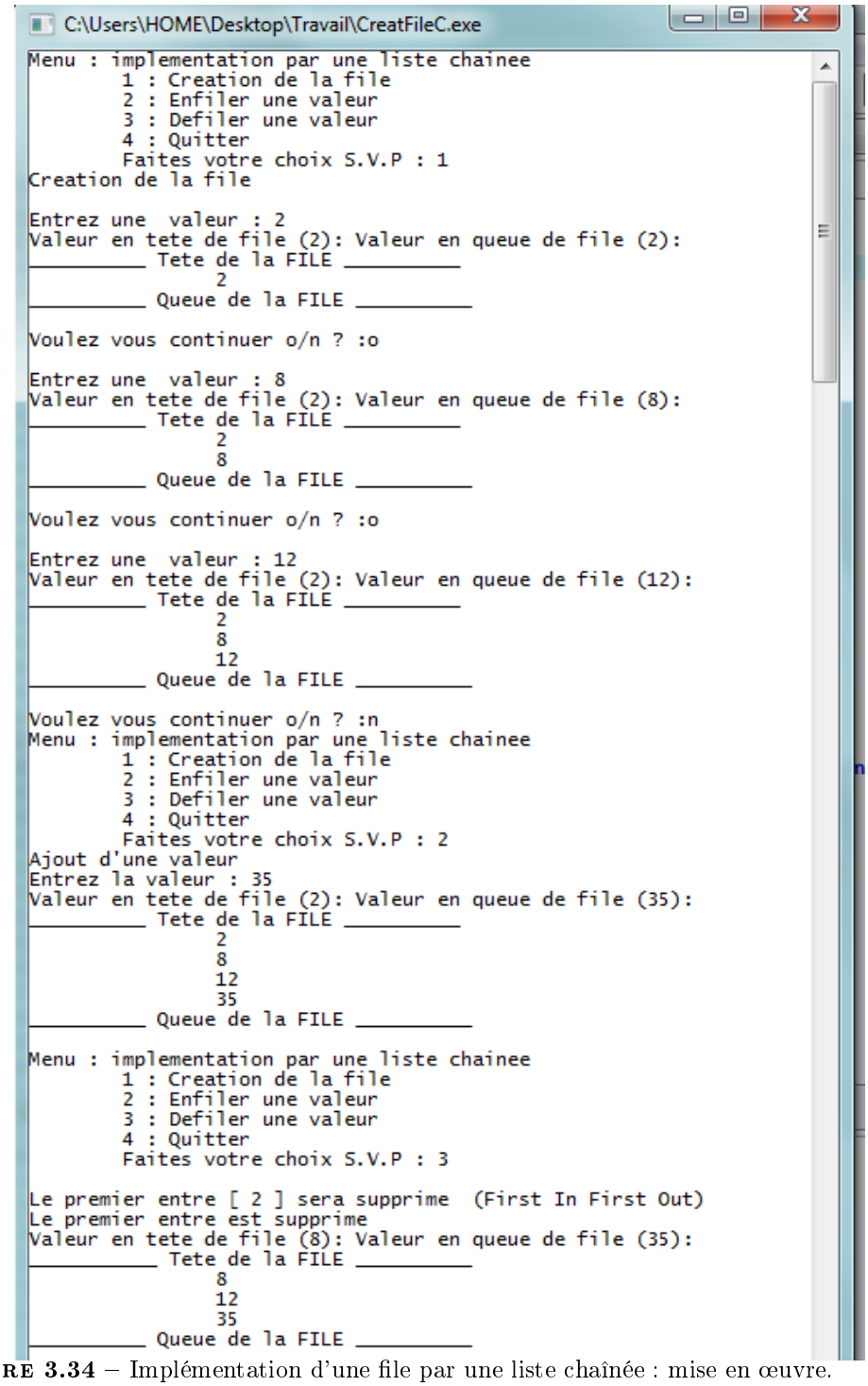


FIGURE 3.34 – Implémentation d'une file par une liste chaînée : mise en œuvre.

En C++, tout comme la structure pile, la structure file fait également partie de la STL (Standard Template Library), il est possible d'y stocker tout type de données (entiers, réels, chaînes de caractères, ...etc). Pour l'utiliser il faut inclure la librairie <queue>, comme suit :

#include <queue> using namespace std ; queue<T> f ; // Où T est le type des éléments de la file f

Principales opérations implémentées :

* **push** : enfiler un élément.
* **pop** : défiler un élément.
* **isEmpty** : teste si la file est vide.
* **isFull** : teste si la file est pleine (implémentation contiguë.
* **front** : retourne l'élément en tête de file sans défilement.
* **back** : retourne l'élément en queue de file.
* **size** : retourne la taille de la file (nombre d'éléments).

☞ Exemple de code :

```
#include<stdio.h> /* Pas nécessaire si on utilise les membres de std cin (lecture) et cout (
    affichage) */
#include <queue>
using namespace std;
queue<float> f; // File de réels
int main(){
printf("Le nombre d'elements de la file est : %d\n",f.size());
printf("Etat de la file : ");
if(f.empty())
   printf("La file est vide\n");
else
 printf("La file n'est pas vide\n");
printf("\nEnfiler 2.5");
f.push(2.5); // Enfiler 2.5
printf("\nEnfiler 5.33");
f.push(5.33); // Enfiler 5.33
printf("\nEnfiler 2");
f.push(2); // Enfiler 2
printf("\nValeur en tete de file = %.2f\n",f.front());
printf("\nValeur en queue de file = %.2f\n",f.back());
// Suppression du premier element
printf("\nDefiler");
f.pop(); // suppression du premier element
printf("\nValeur dde la nouvelle tete = %.2f\n", f.front());
printf("\nDefiler");
f.pop();
printf("\nValeur de la nouvelle tete = %.2f\n", f.front());
}
```

Résultat de l'exécution :

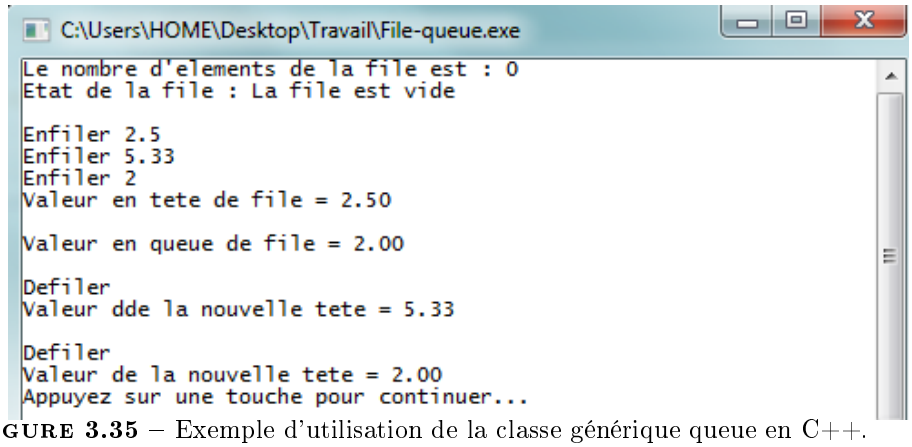


FIGURE 3.35 – Exemple d'utilisation de la classe générique queue en C++.

3.7.2.3 Quelques exemples de traitements sur les files

Les traitements qu'on peut réaliser sur des files sont nombreux, proposons quelques exemples :

☞ Exemple1 : étant donné une file d'entiers et le nombre nb d'éléments qu'elle contient. Ecrire une fonction qui retourne le maximum dans la file.

```
-----------------------------
File f; // f variable globale
int fileVmax(int nb){
int v,max; /*La variable max peut être initialisée à une valeur très grande à retourner si la
    file est vide*/
if(!fileVide(f))
{  defiler(&f,&max); // On suppose à priori le max en tete de file
    enfiler(&f,max);  // Le remettre dans la file
}
// Chercher le vrai max parmi les valeurs
for(i=1;i<=nb;i++)
 {defiler(&f,&v);
   if(v>max)
     max = v;
 enfiler(&f,v);
 }
return max;
}
```

☞ Exemple2 : étant donné une file triée par ordre croissant (plus petite valeur en tête) et une valeur val, on désire insérer val à sa bonne position dans la file f inversée (l'ordre devient décroissant).

```
-----------------------------
File f;
void inverserFile(int val){
Pile p1,p2; // Utilisation de deux piles de travail
{int v;
initPile(p1); initPile(p2);
while(!fileVide(f)&&teteDeFile(f)<val)
  { defiler(&f,&v);
     empiler(&p1,v);
   }
empiler(&p2,val);
while(!fileVide(f))
```

```
     {defiler(&f,&v);
       empiler(&p2,v);
      }
while(!pileVide(p2))
{depiler(&p2,&v);
enfiler(&f,v);
}
while(!pileVide(p1))
{depiler(&p1,&v);
enfiler(&f,v);
}}
```

3.7.3 Résumé

Ce chapitre a introduit la notion d'allocation dynamique de la mémoire précisant son lien étroit avec le type pointeur afin de présenter les structures de données linéaires ou séquentielles de base telles que les listes, les piles et les files qui sont des types abstraits qui pourraient être implémentés (représentés concrètement) à base de tableaux (représentation contiguë) où à base de pointeurs (par des listes chaînées). Chacune des implémentations a ses avantages et ses inconvénients et le choix de l'une ou de l'autre reste à l'initiative du programmeur.

BIBLIOGRAPHIE

.

Amblard, P., J. Fernandez, F. Lagnier, F. Maraninchi, P. Sicard, and P. Waille
2000. *Architectures Logicielles et Matérielles*. Dunod.

Arora, D.
Dernière mise à jour : 12 november 2020. "comprendre la complexité du temps avec des exemples simples". https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/.

Beauquier, D., J. Berstel, and P. Chrétienne
1992. *Éléments d'Algorithmique*. Masson.

Berthet, D. and V. Labatut
2014a. *Algorithmique & programmation en langage C - vol.1 : Supports de cours. Licence. Algorithmique et Programmation*. Istanbul, Turquie. 18, 34

Berthet, D. and V. Labatut
2014b. *Algorithmique & programmation en langage C - vol.2 : Supports de cours. Licence. Algorithmique et Programmation*. Istanbul, Turquie.

Berthet, D. and V. Labatut
2014c. *Algorithmique & programmation en langage C - vol.3 : Corrigés de travaux pratiques. Licence. Algorithmique et Programmation*. Istanbul, Turquie.

Burks, A., H. Goldstine, and J. V. Neumann
1963. *Preliminary discussion of the logical design of an electronic computing instrument*.

Canteaut, A.
. "programmation en langage C". https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/cours.pdf.

Champin, P. A.
. "Listes chainées". http://liris.cnrs.fr/pierre-antoine.champin/enseignement/algo/listes_chainees/.

Cordier, A.
2015. "activité d'introduction à l'algorithmique". https://perso.liris.cnrs.fr/amelie.cordier/teaching/algo/aut2015/IntroAlAlgo.pdf.

Cormen, T.
2010. *Algorithmique*. Dunod.

Cormen, T. H.
2013. *Algorithmes Notions de base*. Collection : Sciences Sup. Dunod.

Cormen, T. H., C. E. Leiserson, and R. L. Rivest
2010. *Algorithmique - Cours avec 957 exercices et 158 problèmes*, 3éme edition. Dunod.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein
2009. *Introduction to Algorithms*, 3rd edition. The MIT Press.

Delannoy, C.
1991. *Apprendre à programmer en Turbo C*. EYROLLES.

Duret-Lutz, A.
Date de publication : 7 novembre 2014. "algorithmique". https://www.coursehero.com/file/69224304/algopdf/.

Edouard, J. M. C.
Date de publication : 07 avril 2008 Dernière mise à jour : 09 janvier 2010. "manipulation des fichiers en c". https://perso.esiee.fr/~landschm/IN3S03/fichiers.pdf.

Froidevaux, C., M. Gaudel, and M. Soria
1990. *Types de données et algorithmes*. McGraw-Hill.

Geerarets, G.
Année Académique 2008–2009 (2^e édition). "Notes du cours FS/1/6584. Année préparatoire au Master en Informatique". https://perso.esiee.fr/~landschm/IN3S03/fichiers.pdf.

Griffiths, M.
1992. *Algorithmique et programmation*. Hèrmes.

Kernighan, B. and D. Richie
1988. *The C programming language*, 2nd edition.

Knuth, D.
1998. *The Art Of Computer Programming(TAOCP)*, 3 edition. Sorting and Searching. Addi–son-Wesley.

Lucas, E.
1892. *Récréations mathématiques*. Librairie Albert Blanchard, tome3. 23

Malgouyres, R., R. Zrour, and F. Feschet
2011. *Initiation à l'algorithmique et à la programmation en C : cours avec 129 exercices corrigés*, 2éme edition. Dunod, Paris. ISBN : 978-2-10-055703-5.

Posamentier, A. and I. Lehmann
2007. *The Fabulous Fibonacci Numbers*, 1 edition. Amherst, N.Y : Prometheus Books. 22

Quercia, M.

2002. *Algorithmique. Cours complet, exercices et problèmes résolus, travaux pratiques.* Vuibert. 18

Raaf, H.

2020. *Algorithmique et Structures de Données 2.* Notes de cours.

Régner, P.

IRIT - Université Paul Sabatier, 2010-2011. "histoire de l'informatique". https://www.irit.fr/~Pierre.Regnier/SitePierre/Enseignement_files/CoursHistoire.pdf.

Renault, E.

Date de publication : 5 octobre 2016. "Introduction à l'algorithmique : notes de cours". https://www.lrde.epita.fr/~renault/teaching/algo/cours.pdf.