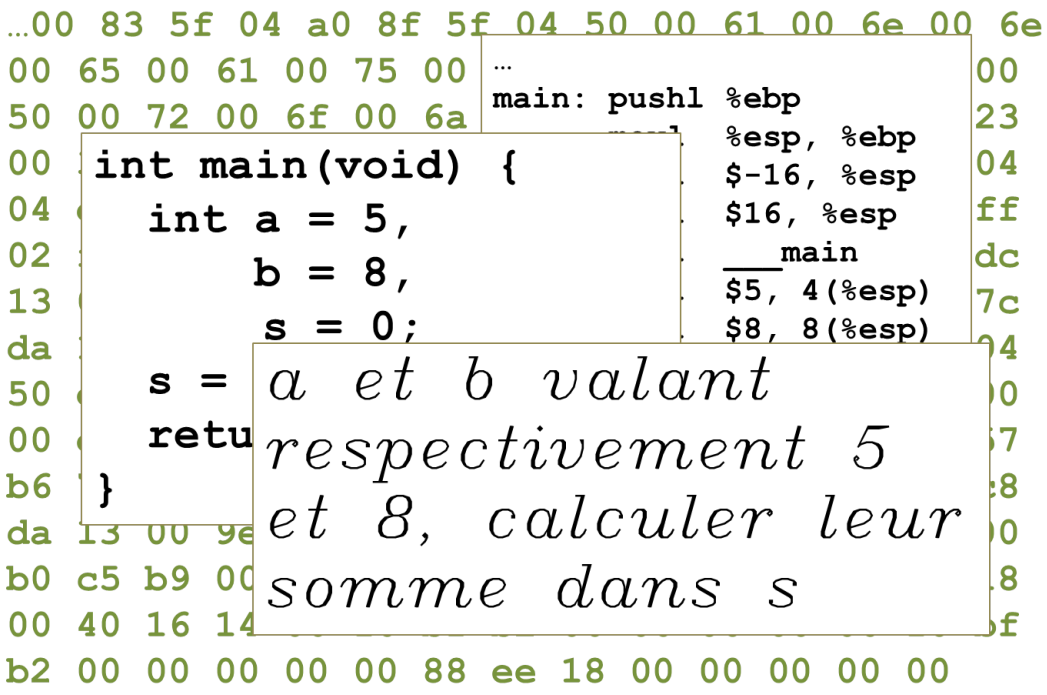
Eléments de cours - Année 2

# Introduction aux structures de données avancées Application aux langages C/C++




1er 2018 - 2019

# Table des matières

# Liste des Algorithmes

# Liste des syntaxes

# Liste des codes sources

# Table des figures

# Liste des tableaux

# Première partie

# Pointeurs

## 1 Pointeurs

Chaque objet mémoire déclaré (donnée, sous-programme) est associé à un espace mémoire qui stocke son contenu. Dans le cas des données, le type détermine la taille de l'espace mémoire réservé : `int` = 4 octets, `char` = 1 octet, etc.

Chaque objet mémoire d'un programme source est transformé en une adresse mémoire (relative au programme) au moment de la compilation [1].

Un objet mémoire aura toujours la même adresse tout au long de l'exécution du programme. L'adresse donne un accès direct à l'objet concerné (adressage direct).

Il est possible de récupérer et d'utiliser l'adresse mémoire d'un objet et de la stocker dans une variable spéciale, le *pointeur* (en anglais : *pointer*).

**Pointeur**

Un *pointeur* est une variable qui peut contenir l'adresse mémoire d'un objet mémoire d'un type donné.

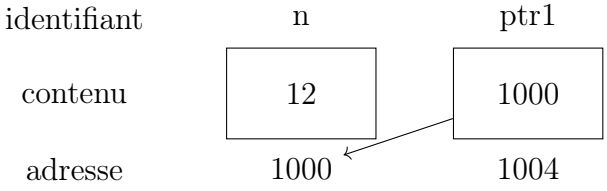
On parle alors d'*adressage indirect* : l'accès au contenu d'un objet mémoire passe par un pointeur qui contient l'adresse de l'objet.

On dit que le pointeur *pointe* (sur) l'objet, fait référence à l'objet pointé.

Les pointeurs sont utilisés essentiellement :

- lors de l'allocation de structures de données dynamiques (tableaux, listes, etc.) dont la taille n'est pas prévisible
- dans le passage des arguments/paramètres aux sous-programmes
  - pour permettre la modification directe des valeurs des arguments (mode de passage par adresse/pointeur en C et C++, mode de passage par référence en C++)
  - pour échanger de grands volumes de données : on transmettra plutôt l'adresse d'un tableau de 10.000 entiers, soit 4 octets, plutôt que la totalité des données (40 ko)

FIGURE 1 – Exemple : la variable n pointée par le pointeur ptr1




La figure 2 (page 2) décrit un extrait d'une mémoire contenant 4 variables :

1. Cette adresse est translatée au moment du chargement en mémoire vive en vue de l'exécution du programme

— *a* de type `char` : elle occupe 1 octet

— *ptr1* de type pointeur sur `char` : elle occupe 4 octets (remarquer le décalage de 3 octets pour aligner le pointeur sur une frontière de mot, groupe de 4 octets en général)

— *i* de type `int` : elle occupe 4 octets

— *ptr2* de type pointeur sur `int` : elle occupe 4 octets

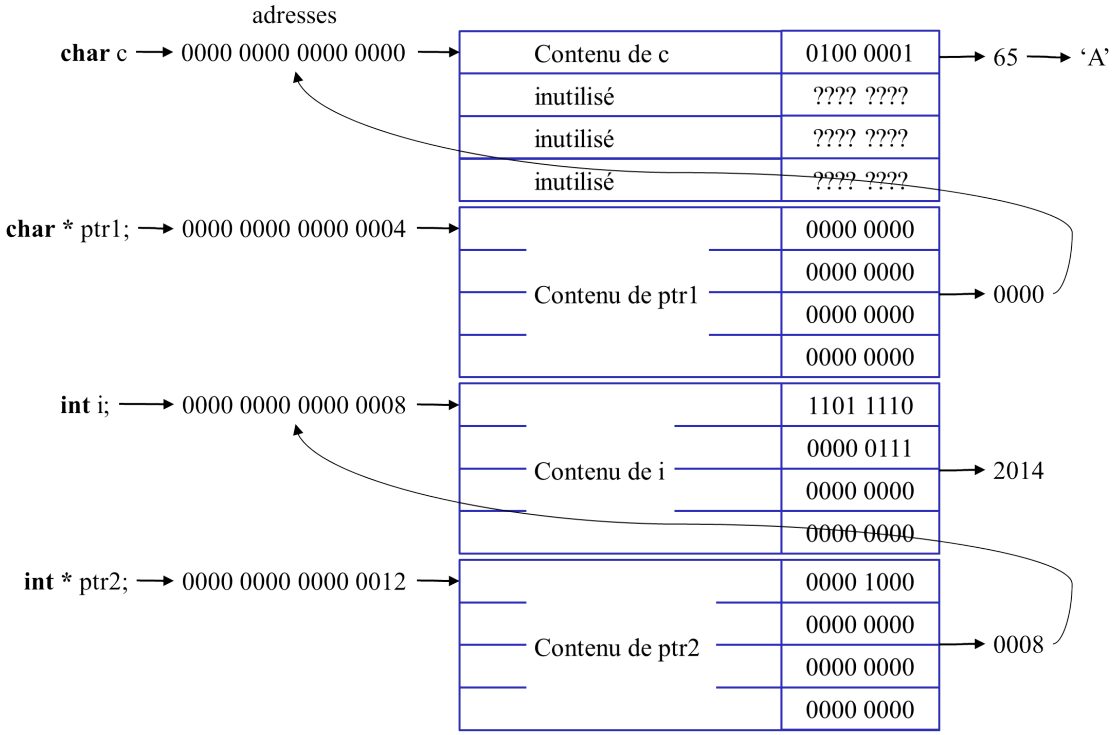
Le contenu des pointeurs nous permet de dire que *ptr1 pointe sur a* et *ptr2 pointe sur i*

Un pointeur étant une variable, il occupe généralement un espace mémoire d'un *mot machine* (en anglais : *word*) : sa taille dépend de l'architecture matérielle, mais aussi du compilateur utilisé et du type de pointeur souhaité.

On peut la connaitre en utilisant la fonction `sizeof` : **sizeof**(T $*$) où `T` est le type de donnée pointée (4 octets sur les architectures 32 bits).

La taille occupée par un pointeur ne dépend pas du type de donnée pointée.

FIGURE 2 – variable n pointée par le pointeur ptr1




## 1.1 Déclarer un pointeur

Comme toute autre variable, un pointeur doit être déclaré avant son utilisation.

**Syntaxe 1.1 Déclarer un pointeur C/C++**

```
T * idPtr ;
```

où :

— T : type de variable pointée

— * : opérateur signifiant "pointeur vers une variable du type T"

— idPtr : identifiant donné au pointeur.

Remarque : le type de donnée pointée permet au compilateur de connaitre la taille de l'objet pointé : elle est utilisée dans les tableaux pour passer d'un élément pointé vers le suivant en mémoire (voir section 1.5.2 page 7).

Code source 1 – Exemples de déclaration de pointeurs en C

```
1 int * p1; // p1 est un pointeur vers un entier
2 float * p2; // p2 est un pointeur vers un réel
3 char * p3; // p3 est un pointeur vers un caractère
```

**Conseil**

Si un pointeur n'est pas utilisé immédiatement, il est souhaitable de l'initialiser à la valeur `NULL` :

Code source 2 – Exemples de déclaration de pointeurs en C

```
1 int * p1 = NULL; // p1 est un pointeur vers un entier
2 float * p2 = NULL; // p2 est un pointeur vers un réel
3 char * p3 = NULL; // p3 est un pointeur vers un caractère
```

**Attention**

```
1 int * p1, p2;
```

déclare :

— p1 : comme pointeur vers entier

— p2 : comme variable de type entier

et non pas 2 pointeurs !

## 1.2 Adressage : opérateur &

L'`adressage` est l'opération de récupération de l'adresse d'un objet mémoire.

**Syntaxe 1.2 Opérateur d'adressage C/C++**

& id

où :

— & : opérateur d'extraction de l'adresse mémoire de l'objet qui suit

— id : identifiant d'un objet mémoire déclaré

L'adresse mémoire extraite par l'opérateur d'adressage peut être stockée dans un pointeur :

Code source 3 – Utilisation de l'adressage en C

```
1 int x = 10; // x est un entier
2 int * p = NULL; // p est un pointeur vers un entier
3 p = &x; // p contient l'adresse de l'entier x
```

## 1.3 Déréférencement : opérateur *

Le `déréférencement` (en anglais : *indirection, dereference*) est l'opération de récupération indirecte du contenu d'une donnée à partir de la valeur d'un pointeur (une adresse pointée).

**Syntaxe 1.3 opérateur de déréférencement C/C++**

* idPtr

où :

- `*` : opérateur de déréférencement
- *idPtr* : identifiant du pointeur

Code source 4 – Exemple d'utilisation d'un pointeur

```
12          /* DECLARATIONS, INITIALISATIONS */
13          int x = 10;
14          int * p = NULL;
15          p = &x;
16          printf("\nAdresse de x = %p, contenu de x = %d", &x, x);
17          printf("\nAdresse de x = %p, contenu de x = %d", p, *p);
18
19          /* TRAITEMENT */
20      /* ajouter 2 à x : adressage direct */
21          x = x + 2;
22      /* ajouter 2 à x : adressage indirect */
23          *p = *p + 2;
24            /* afficher le contenu de x */
25          printf("\nContenu de x = %d, contenu de x = %d", x, *p);
```

Résultat de l'exécution :

```
Adresse de x = 0028ff18, contenu de x = 10
Adresse de x = 0028ff18, contenu de x = 10
Contenu de x = 14, contenu de x = 14
```

## 1.4 Pointeurs et enregistrements

Les pointeurs peuvent être utilisé pour pointer sur une variable de type enregistrement.

### 1.4.1 Accès aux membres

L'accès indirect aux membres, à partir d'un pointeur,

**Syntaxe 1.4 accès indirect aux membres C/C++**

```
(* idPtr).member
     /* ou bien */
idPtr -> idMemb
```

où :

- `*` : opérateur de déréférencement
- `->` : opérateur d'accès à un membre d'une variable enregistrement pointée
- *idPtr* : identifiant du pointeur
- *idMemb* : identifiant d'un membre

**Attention**

L'opérateur . (accès à un membre) a une priorité supérieure à celle de * (déréférencement), d'où la nécessité d'encadrer le déréférencement par des parenthèses

### 1.4.2 Exemple

Code source 5 – Enregistrement Point en C

```
10 /* définition d'un point */
11 struct Point
12 {
13         float x, y;
14 };
15 /* déclaration d'un alias sur l'enregistrement Point */
16 typedef struct Point Point;
```

Code source 6 – Déclaration d'un pointeur sur enregistrement

```
20         /* DECLARATIONS, INITIALISATIONS */
21   Point p1; // déclaration d'une variable de type Point
22   Point * ptr; // déclaration d'un pointeur vers le type Point
23   ptr = &p1; // ptr contien maintenant l'adresse de p1
```

Code source 7 – Accès direct aux membres d'un enregistrement

```
25         /* TRAITEMENT */
26     /* saisie sans pointeur */
27   printf("Saisie 1 : ");
28   scanf("%f %f", &p1.x, &p1.y);
29     /* affichage sans pointeur */
```

```
30    printf("%f %f, adresses : %p %p %p\n",
31          p1.x, p1.y, &p1, &(p1.x), &(p1.y));
```

```
Saisie 1 : 1 2
1.000000 2.000000, adresses : 0028ff14 0028ff14 0028ff18
```

Code source 8 – Accès indirect aux membres d'un enregistrement

```
33      /* saisie avec pointeur */
34    printf("Saisie 2 : ");
35    scanf("%f %f", &(*ptr).x, &(*ptr).y);
36      /* affichage avec pointeur */
37    printf("%f %f \n", (*ptr).x, (*ptr).y);
```

```
Saisie 2 : 3 4
3.000000 4.000000
```

Code source 9 – Accès indirect aux membres d'un enregistrement par pointeur sur le membre

```
39      /* affichage avec pointeur */
40    printf("Saisie 3 : ");
41    scanf("%f %f", &(ptr->x), &(ptr->y));
42      /* affichage avec pointeur */
```

```
Saisie 3 : 5 6
5.000000 6.000000
```

## 1.5 Pointeurs et tableaux

La déclaration d'un tableau réserve un espace mémoire contigu nécessaire à l'ensemble de ses éléments : elle associe l'adresse de début de cet espace, soit celle le premier élément du tableau, au tableau déclaré.

### 1.5.1 Déclaration d'un pointeur vers tableau

Déclarer un pointeur vers un tableau d'un certain type consiste à déclarer un pointeur du même type que le tableau et vers le 1er élément du tableau ; on pourra effectuer des opérations de décalage d'adresse afin d'accéder aux différents éléments du tableau.

**Syntaxe 1.5 accès indirect aux membres C/C++**

```
T idTab[size];
T * idPtr = NULL;
idPtr = idTab;
    /* ou bien : */
```

```
idPtr = &idTab[0];
```

où :

- `T` : type de donnée
- *idTab* : identifiant du tableau
- *size* : nombre d'éléments
- `T *` : pointeur vers T
- *idPtr* : identifiant du pointeur
- &idTab[0] : adresse du 1er élément du tableau

Remarque : en C++, std :: array représente une forme de structure qui contient un tableau de taille fixe. On ne pourra pas utiliser un pointeur vers le tableau qui puisse accéder à ses différents éléments.

### 1.5.2 Arithmétique des pointeurs

L'accès aux éléments alloués d'un tableau utilise l'adresse de départ (celle du 1er élément) en lui ajoutant uen valeur de décalage : 0 pour le 1er, 1 pour le 2nd, etc.

Le fait d'incrémenter une adresse de pointeur donne accès à l'adresse de l'élément suivant, et correspond à un décalage mémoire d'un nombre d'octets correspondant à la taille du type de donnée pointée.

**Syntaxe 1.6 Opération sur pointeur C/C++**

```
* (idPtr + shift);
```

- `*` : l'opérateur de déréférencement
- *idPtr* : identifiant du pointeur
- *shift* : décalage effectué (nombre entier)

Par exemple, **int** tab[4] réserve un espace de 4 x 4 octets soit 16 octets et associe l'adresse de début de cet espace, soit celle le premier élément du tableau, à la variable *tab*.

Code source 10 – Accès aux éléments d'un tableau

```
12          /* DECLARATIONS, INITIALISATIONS */
13    int tab[] = {16, 29, 6, 1}; // déclaration d'un tableau
14    int * ptr; // déclaration d'un pointeur vers entier
15    ptr = tab; // ptr contient maintenant l'adresse de tab
16     /* ou bien : ptr = &tab[0]; // (de son 1er élément) */
17
18          /* TRAITEMENT */
19      /* affichage de l'adresse de début de tab */
20    printf("\nAdresse de debut de tableau (a) : %p", tab);
21    printf("\nAdresse de debut de tableau (b) : %p", &tab[0]);
22    printf("\nAdresse de debut de tableau (c) : %p", ptr);
```

Résultat :

```
Adresse de debut de tableau (a) : 0028fef0
Adresse de debut de tableau (b) : 0028fef0
Adresse de debut de tableau (c) : 0028fef0
```

Code source 11 – Accès aux éléments d'un tableau

```
23
24      /* affichage des adresses de chacun des éléments */
25    printf("\n");
26    for (int i = 0; i < 4; ++i) {
27      printf("\nAdresse de l'element %d : %p", i, &tab[i]);
28    }
```

```
Adresse de l'element 0 : 0028fef0
Adresse de l'element 1 : 0028fef4
Adresse de l'element 2 : 0028fef8
Adresse de l'element 3 : 0028fefc
```

Code source 12 – Accès aux éléments d'un tableau

```
22    printf("\nAdresse de debut de tableau (c) : %p", ptr);
23
24      /* affichage des adresses de chacun des éléments */
25    printf("\n");
26    for (int i = 0; i < 4; ++i) {
27      printf("\nAdresse de l'element %d : %p", i, &tab[i]);
28    }
29      /* affichage des adresses de chacun des éléments */
30    printf("\n");
31    for (int i = 0; i < 4; ++i) {
32      printf("\nAdresse de l'element %d : %p", i, ptr+i);
33    }
```

```
Adresse de l'element 0 : 0028fef0
Adresse de l'element 1 : 0028fef4
Adresse de l'element 2 : 0028fef8
Adresse de l'element 3 : 0028fefc
```

Code source 13 – Accès aux éléments d'un tableau

```
34
35      /* affichage des valeurs de chacun des éléments */
36    printf("\n");
37    for (int i = 0; i < 4; ++i) {
38      printf("%d ", tab[i]);
39    }
40      /* affichage des valeurs de chacun des éléments */
41    printf("\n");
42    for (int i = 0; i < 4; ++i) {
```

```
43      printf("%d ", *(ptr+i));
44    }
45      /* modification des valeurs de chacun des éléments */
46    for (int i = 0; i < 4; ++i) {
47      tab[i]++;
48    }
49      /* modification des valeurs de chacun des éléments */
50    for (int i = 0; i < 4; ++i) {
51      (*(ptr+i))++;
52    }
53      /* affichage des valeurs de chacun des éléments */
54    printf("\n");
55    for (int i = 0; i < 4; ++i) {
56      printf("%d ", tab[i]);
57    }
```

```
16 29 6 1
16 29 6 1
18 31 8 3
```

### 1.5.3 Tableaux d'enregistrement

Pour accéder aux membres d'un des enregistrements d'un tableau, on devra préciser l'adresse de élément concerné :

**Syntaxe 1.7 Accès aux membres d'un tableau d'enregistrements C/C++**

```
T idEnreg[N];
T * idPtr;
idPtr = idEnreg;
/* ou idPtr = idenreg[0]*/
...
(*(idPtr + shift)).membre
     /* ou bien */
(idPtr + i)->membre
```

Code source 14 – Accès aux membres d'un tableau d'enregistrements : adresses

```
15 {
16          /* DECLARATIONS, INITIALISATIONS */
17    Point tab[] = {{1,2}, {3,4}}; // déclaration d'un tableau
18    Point * ptr; // déclaration d'un pointeur vers Point
19    ptr = tab; // ptr contien maintenant l'adresse de tab
20     /* ou bien : ptr = &tab[0]; // (de son 1er élément)*/
21
```

```
22          /* TRAITEMENT */
23      /* affichage des adresses de chacun des éléments */
24    printf("\n");
25    for (int i = 0; i < 2; ++i) {
26      printf("\nAdresse de l'element %d : %p", i, &tab[i]);
27    }
28      /* affichage des adresses de chacun des éléments */
29    printf("\n");
30    for (int i = 0; i < 2; ++i) {
31      printf("\nAdresse de l'element %d : %p", i, ptr+i);
32    }
```

```
Adresse de l'element 0 : 0028fefc
Adresse de l'element 1 : 0028ff04

Adresse de l'element 0 : 0028fefc
Adresse de l'element 1 : 0028ff04
```

Code source 15 – Accès aux membres d'un tableau d'enregistrements : valeurs

```
33
34      /* affichage des valeurs de chacun des éléments */
35    printf("\n\n");
36    for (int i = 0; i < 2; ++i) {
37      printf("(%f,%f) ", tab[i].x, tab[i].y);
38    }
39      /* affichage des valeurs de chacun des éléments par pointeur */
40    printf("\n");
41    for (int i = 0; i < 2; ++i) {
42      printf("(%f,%f) ", (*(ptr+i)).x, (*(ptr+i)).y);
43    }
44      /* affichage des valeurs de chacun des éléments par pointeur */
45    printf("\n");
46    for (int i = 0; i < 2; ++i) {
47      printf("(%f,%f) ", (ptr+i)->x, (ptr+i)->y);
48    }
```

```
(1.000000,2.000000) (3.000000,4.000000)
(1.000000,2.000000) (3.000000,4.000000)
(1.000000,2.000000) (3.000000,4.000000)
```

### 1.5.4 Chaines de caractères

Une chaine de caractères peut être définie comme un tableau de caractères[2]. On peut donc définir un pointeur vers le tableau.

2. en C, c'est la seule manière de définir une chaine, alors que le C++ offre également le type `string` qui est un objet qui encapsule un tableau dynamique de caractères

Code source 16 – Déclaration d'une chaine (à la manière du langage C)

```
11 {
12         /* DECLARATIONS, INITIALISATIONS */
13   char ch[] = "Bye"; // déclaration de la chaine
14   char * ptr; // déclaration d'un pointeur vers entier
15   ptr = ch; // ptr contien maintenant l'adresse de ch (de son 1er élément)
16    /* ou bien : ptr = &ch[0] */
```

Code source 17 – Affichage de l'adresse de début de la chaine

```
19     /* affichage de l'adresse de début de la chaine */
20   printf("\nAdresse de debut de chaine 1 : %p", ch);
21   printf("\nAdresse de debut de chaine 2 : %p", &ch[0]);
```

```
Adresse de debut de chaine 1 : 0028ff08
Adresse de debut de chaine 2 : 0028ff08
Adresse de debut de chaine 3 : 0028ff08
```

Code source 18 – Affichage de l'adresse de début de la chaine

```
23
24     /* affichage des adresses de chacun des éléments */
25   printf("\n");
26   for (unsigned int i = 0; i < (sizeof ch /sizeof(char)); ++i) {
27     printf("\nAdresse du caractere %d : %p", i, &ch[i]);
28   }
29     /* affichage des adresses de chacun des éléments */
30   printf("\n");
31   for (unsigned int i = 0; i < (sizeof ch /sizeof(char)); ++i) {
32     printf("\nAdresse du caractere %d : %p", i, ptr+i);
33   }
```

```
Adresse du caractere 0 : 0028ff08
Adresse du caractere 1 : 0028ff09
Adresse du caractere 2 : 0028ff0a
Adresse du caractere 0 : 0028ff08
Adresse du caractere 1 : 0028ff09
Adresse du caractere 2 : 0028ff0a
Adresse du caractere 3 : 0028ff0b
Adresse du caractere 3 : 0028ff0b
```

Code source 19 – Affichage du contenu de chaque élément d'une chaine

```
34
35     /* affichage des valeurs de chacun des éléments */
36   printf("\n");
37   for (unsigned int i = 0; i < (sizeof ch /sizeof(char)); ++i) {
38     printf("%c/%d ", ch[i], ch[i]);
```

```
39    }
40      /* affichage des valeurs de chacun des éléments */
41    printf("\n");
42    for (unsigned int i = 0; i < (sizeof ch /sizeof(char)); ++i) {
43      printf("%c/%d ", *(ptr+i), *(ptr+i));
44    }
```

```
B/66 y/121 e/101  /0
B/66 y/121 e/101  /0
```

## 1.6 Pointeurs et passage des paramètres par adresse

Dans le mode de passage par adresse, le sous-programme définit un paramètre comme pointeur. L'appelant devra donc lui passer l'adresse d'une variable (ou constante) déclarée.

Code source 20 – Passage par adresse en C/C++

```
1 void doubler(int * n)
2 {
3         *n = *n * 2;
4 }
5 int main()
6 {
7         int i = 5;
8         doubler(&i);
9 }
```

Remarque : en C++, on tirera profit du mode de passage par référence (alias vers l'argument) qui rend le mode de passage par adresse transparent.

Code source 21 – Passage par référence en C++

```
1 void doubler(int & n)
2 {
3         n = n * 2;
4 }
5 int main()
6 {
7         int i = 5;
8         doubler(i);
9 }
```

## 1.7 Pointeurs sur fonctions

Les pointeurs peuvent également faire référence à des sous-programmes qui sont également des objets mémoire et qui possèdent une adresse d'appel.

### 1.7.1 Déclaration d'un pointeur sur fonction

Code source 22 – Définir une fonction et un pointeur vers fonction

```
1  int somme(int n1, int n2)
2  {
3    return (n1 + n2) ;
4  };
5
6  int (*ptrf)(int, int); // decl. pointeur sur fonction
```

Le pointeur *ptrf*

- est de type `int`, le type de la fonction
- et définit 2 paramètres `int`, ceux de la fonction.

### 1.7.2 Initialisation d'un pointeur sur fonction

```
1  int (*ptrf)(int, int); // decl. pointeur sur fonction
2  ptrf = &somme; // le pointeur pointe sur la fonction fSomme
```

### 1.7.3 Appel d'un fonction par pointeur

Le pointeur peut être utilisé en utilisant l ?opérateur * (opérateur de déréférencement) pour acvéder à la fonction :

```
1  int a = 10;
2  int b = 20;
3  int c = (*ptrf)(a, b);
```

La variable c reçoit la valeur retournée par l'appel de la fonction pointée par ptrf à laquelle les valeurs de a et b ont été passées

Exemple association tableau de pointeurs sur fonctions :

```
1  // définition de fonctions de calculs arithmétiques
2  int somme(int n1, int n2)
3   {  return (n1 + n2) ; }
4  int difference(int n1, int n2)
5   {  return (n1 - n2) ;}
6  int produit(int n1, int n2)
7   {  return (n1 * n2) ; }
8  int quotient(int n1, int n2)
9   {  return (n1 / n2) ;}
10 int modulo(int n1, int n2)
11  {  return (n1 % n2) ;}
12
13 typedef int (*ptrf)(int, int); // décl. pointeur sur fonction
14
15
16 int main (void)
17 {
18   int a, b, c ;
```

```
19    ptrf tabFonc[5] ; // allocation d'un tableau de pointeurs
20    tabFonc[0]= &somme ;
21    tabFonc[1]= &difference ;
22    tabFonc[2]= &produit ;
23    tabFonc[3]= &quotient ;
24    tabFonc[4]= &modulo ;
25
26    cout << "entrez 2 nombres " ;
27    cin >> a >> b ;
28
29    cout << " entrez un numéro de fonction (0 à 4) " ;
30    cin >> c;
31
32    cout << " resultat = " << (*(tabFonc[c]))(a, b) );
33
34    return 0 ;
35  }
```

### 1.7.4 Exemple 2

```
1  void afficherTableau(int t[], int nb, void (*f)(int)) {
2    int i;
3    for (i = 0; i < nb; i++) f(t[i]);
4  }
5
6  void afficherNombre1(int n) {
7    printf("%d ", n);
8  }
9  void afficherNombre2(int n) {
10   printf("%d\n", n);
11 }
12
13 int main()
14 {
15         int t[12];
16         for(i = 0;i<12;++i) {
17                 t[i] = i*2;
18         }
19   afficherTableau(t, 12, afficherNombre1);
20   afficherTableau(t, 12, afficherNombre2);
21 }
```

Une fonction passée en argument à une autre fonction est appelée fonction de rappel (en anglais : *callback*). A voir.

## 1.8 Double adressage par pointeur

On peut utiliser des pointeurs contenant les adresses d'autres pointeurs, eux-mêmes pointant vers des objets mémoire.

**Syntaxe 1.8 Pointeur vers pointeur C/C++**

```
T ** idPtr;
```

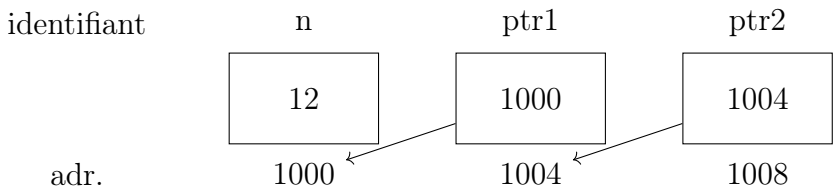
où :

— `T **` : pointeur vers pointeur sur type T

— *idPtr* : identifiant du pointeur

Code source 23 – exemple de pointeur sur pointeur sur entier

```
1  /* déclarations */
2  int n = 12;
3  int * ptr1;
4  int ** ptr2;
5  /* initialisation */
6  ptr1 = & n;
7  ptr2 = & ptr1;
8  /* affichage de la valeur de n */
9  printf("n vaut = %d\n", n );
10 printf("n vaut = %d\n", *ptr1 );
11 printf("n vaut = %d\n", **ptr2 );
```




L'exemple suivant, crée un pointeur vers un tableau de pointeur sur T, chacun pointant sur un espace alloué :

**Syntaxe 1.9 Pointeur vers tableau de pointeurs (alloc.dyn.) C/C++**

```
T ** idPtr = (T **) malloc(sizeof(T *) * n);
idPtr[0] = (T *) malloc(sizeof(T));
idPtr[1] = (T *) malloc(sizeof(T));
...
```

où :

— `T **` : pointeur vers pointeur sur type T

— *idPtr* : identifiant du pointeur

— $n$ : nombre d'éléments du tableau de pointeurs

Code source 24 – Pointeur sur pointeur sur caractères

**Exemple d'utilisation pour un tableau de chaines**

```
10
11 int main()
12 {
13   char **pp;
14   char * p;
15   int i;
16   pp = (char **)malloc(200);
17   pp[0] = (char *)"Janvier";    // *pp pointe vers 'J', *(pp+1) pour vers 'a'
18   pp[1] = (char *)"Fevrier";
19   pp[2] = (char *)"Mars";
20   pp[3] = (char *)"Avril";
21   pp[4] = (char *)"Mai";
22
23   for (i=0; i < 4; ++i) {
24       /* (affiche de l'adr. debut jusqu'à '\0' */
25     printf("\n%s (%p)", *(pp+i), (pp+i));
26   }
27   printf("\n");
28   for (i=0; i < 4; ++i) {
29     p = *(pp+i);
30     printf("\nadr.debut (%p) ", p);
31     do {
32       printf("\n\t%c (%p),", *(p), p);
33       ++p;
34     } while (*p != '\0');
35     printf("\n");
36   }
37 }
```

Exécution :

```
Janvier (00771728)
Fevrier (0077172c)
Mars (00771730)
Avril (00771734)
adr.debut (00408024)
        J (00408024),
        a (00408025),
        n (00408026),
        v (00408027),
        i (00408028),
        e (00408029),
        r (0040802a),

adr.debut (0040802c)
        F (0040802c),
        e (0040802d),
        v (0040802e),
```

```
            r (0040802f),
            i (00408030),
            e (00408031),
            r (00408032),

    adr.debut (00408034)
            M (00408034),
            a (00408035),
            r (00408036),
            s (00408037),

    adr.debut (00408039)
            A (00408039),
            v (0040803a),
            r (0040803b),
            i (0040803c),
            l (0040803d),
```

Un exemple d'utilisation de pointeur vers pointeur : lorsqu'on souhaite modifier l'adresse d'un pointeur passé à une fonction :

Code source 25 – Tentative de modifier l'adresse pointée : sans effet

```
9
10  void modifierSansEffet(int * a, int * b)
11 {
12         printf("\n\tavant a: %d (%p), b: %d (%p)", *a, a, *b, b);
13         a = b;
14         printf("\n\tapres a: %d (%p), b: %d (%p)", *a, a, *b, b);
15 }
16
17 int main(void)
18 {
19         /* DECLARATIONS, INITIALISATIONS */
20         int a = 5,
21                 b = 6;
22   int * pa = &a;
23   int * pb = &b;
24
25         /* TRAITEMENT */
26     /* modification des contenu puis
27        affichage de l'adresse de début de la chaine */
28         printf("appel a: %d (%p), b: %d (%p)", *pa, pa, b, &b);
29         modifierSansEffet(pa, pb);
30         printf("\nretour a: %d (%p), b: %d (%p)", *pa, pa, b, &b);
```

Résultat :

```
appel  a: 5 (0028ff14), b: 6 (0028ff10)
        avant a: 5 (0028ff14), b: 6 (0028ff10)
        apres a: 6 (0028ff10), b: 6 (0028ff10)
retour a: 5 (0028ff14), b: 6 (0028ff10)
```

**Attention**

Les pointeurs sont passés par valeur : on peut accéder ou modifier le contenu des valeurs pointées mais la modification des adresses (la valeur du pointeur) est locale.

L'utilisation d'un pointeur sur pointeur permet cette modification :

Code source 26 – Tentative de modifier l'adresse pointée : avec effet

```
9
10  void modifierAvecEffet(int ** a, int * b)
11 {
12         printf("\n\tavant a: %d (%p), b: %d (%p)", **a, *a, *b, b);
13         *a = b;
14         printf("\n\tapres a: %d (%p), b: %d (%p)", **a, *a, *b, b);
15 }
16
17 int main(void)
18 {
19         /* DECLARATIONS, INITIALISATIONS */
20         int a = 5,
21                 b = 6;
22   int * pa = &a;
23   int * pb = &b;
24   int ** ppa = &pa;
25
26         /* TRAITEMENT */
27     /* modification des contenu puis
28        affichage de l'adresse de début de la chaine */
29         printf("appel a: %d (%p), b: %d (%p)", *pa, pa, b, &b);
30         modifierAvecEffet(ppa, pb);
31         printf("\nretour a: %d (%p), b: %d (%p)", *pa, pa, b, &b);
```

Résultat :

```
appel  a: 5 (0028ff14), b: 6 (0028ff10)
        avant a: 5 (0028ff14), b: 6 (0028ff10)
        apres a: 6 (0028ff10), b: 6 (0028ff10)
retour a: 6 (0028ff10), b: 6 (0028ff10)
```

Le pointeur *pa* pointe maintenant vers *b*.

## 1.9 Risques induits par l'utilisation des pointeurs

**Attention**

L'usage des pointeurs est peu contrôlé par les compilateurs. Une erreur dans l'initialisation d'un pointeur conduit souvent à une erreur d'exécution car on risque d'accéder à des espaces hors des limites du programme. Mais une erreur dans l'initialisation d'un

pointeur peut masquer un dysfonctionnement : le programme fonctionne apparemment mais accède à des données qui ne sont pas celles attendues...

## 1.10 Pointeurs et allocation dynamique

Une déclaration statique pour traiter un flux aléatoire comporte un risque :

— *faible*, sans conséquence, si le nombre d'éléments réservés est sous-utilisé,

— *grave*, si le nombre de données du flux de données à traiter dépasse les zones mémoire réservées pour ces éléments : on a alors un débordement (en anglais : *overflow*). Il y aura soit des éléments non traités soit plantage du programme.

L'allocation dynamique consiste, pendant l'exécution d'un programme,

— à demander au système d'exploitation l'allocation d'un espace pouvant contenir un certain nombre de valeurs d'un certain type de données,

— et à récupérer, dans une variable pointeur, l'adresse de la 1ère case allouée (similaire à un tableau).

Elle répond au besoin de réserver un espace mémoire en fonction des besoins du programme en cours d'exécution.

**Attention**

Chaque demande d'allocation dynamique devra faire l'objet d'une demande de libération de l'espace alloué (voir section 1.10.2 page 22).

### 1.10.1 Allocation dynamique d'espace

**1.10.1.1 Fonction `malloc` en C/C++** L'entête de la fonction malloc est déclarée dans le fichier d'entête <stdlib.h>. Le prototype de la fonction est :

**Syntaxe 1.10 fonction malloc : prototype**

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

où :

— void * : type retourné par la fonction, pointeur générique

— `malloc` : fonction d'allocation

— size_t n : n, nombre d'octets demandé, size_t, type de n, entier naturel

Ainsi, pour réserver un espace mémoire :

**Syntaxe 1.11 Allocation dynamique**

```
T * idPtr = malloc(n);
/* ou, si le pointeur est déjà déclaré :*/
```

```
idPtr = (T *) malloc(n);
```

où :

- `T *` : type de donnée pointé (celui qui sera réservé)
- *idPtr* : identifiant du pointeur
- `malloc` : fonction d'allocation
- `(T *)` : transtypage de `(void *)` vers `(T *)` (la fonction `malloc` renvoie un pointeur générique, `void *` : le transtypage est parfois nécessaire)
- *n* : nombre d'octets demandé qui devra tenir compte de la taille du type de donnée pointée, d'où l'utilisation de la fonction `sizeof`(T) qui renvoie le nombre d'octets utilisé par le type T

Code source 27 – Allocation dynamique : malloc

```
1 // allocation dynamique d'un entier
2 int * idPtr = malloc(sizeof(int));
3 // allocation dynamique de 10 entiers
4 int * idPtr = malloc(sizeof(int) * 10);
```

L'utilisation du pointeur *idPtr* est ensuite identique à celui d'une valeur unique ou d'un tableau de 10 nombres.

Il se peut que la demande de réservation d'espace échoue, par exemple dans le cas où l'espace mémoire disponible n'est pas suffisant pour répondre à la demande. Il est important de prévoir ce cas et de contrôler que le pointeur contient bien une valeur retournée par `malloc` différente de `NULL`

Code source 28 – Allocation dynamique et gestion d'erreur

```
12 {
13         /* DECLARATIONS, INITIALISATIONS */
14   int n = 0;
15   int i = 0;
16   int * idPtr = NULL;
17         /* TRAITEMENT */
18     /* demander le nombre de valeurs */
19   printf("Nombre de valeurs ? ");
20   scanf("%d", &n);
21   while (n < 1) {
22     printf("\nErreur : doit être strictement positif");
23     printf("\nNombre de valeurs ?");
24     scanf("%d", &n);
25   }
26         /* TRAITEMENT */
27     /* demande l'allocaion d'espace mémoire */
28   idPtr = malloc(n * sizeof(int));
29   if (idPtr == NULL) {
30     printf("erreur allocation mémoire");
31     exit(EXIT_FAILURE);
32   }
33   printf("\nTaille de l'espace alloué : %d", n * sizeof (int));
```

```
34
35      /* initialisation et affichage des valeurs */
36    printf("\n");
37    for (i = 0; i < n; ++i) {
38      idPtr[i] = i+1; /* ou *(idPtr + i) = i+1 */
39      printf("%d ", idPtr[i]);
40    }
```

L'exécution produit le code suivant :

```
Nombre de valeurs ? 24
Taille de l'espace allouÚ : 96
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

On peut remarquer :

— le test qui suit la demande d'allocation mémoire

— la gestion de l'espace comme celui d'un tableau

```
1 int * ptr ;
2 ptr = new int() ;
3 *ptr=10 ;
4
5 char * mot ;
6 mot = new char[20] ;
```

Deux autres fonctions d'allocation permettent

— l'initialisation de l'espace alloué : `calloc`

— l'extension (ou la réduction) d'un espace déjà alloué tout en préservant les données déjà mémorisées : `realloc`

**1.10.1.2 Opérateur `new` en C++** L'opérateur `new` permet l'allocation dynamique d'un nouvel espace mémoire :

**Syntaxe 1.12 opérateur C++ new : prototype**

```
void * operator new ( std : :size_t count ) ;
void * operator new[]( std : :size_t count ) ;
```

où :

— void * : type retourné par l'opérateur, pointeur générique

— `operator new` : l'opérateur new

— `std::size_t` *count* : count, nombre d'octets demandé, size_t, type de n, entier naturel

—

: allocation d'un tableau

Ainsi, pour réserver un espace mémoire :

**Syntaxe 1.13 Allocation dynamique en C++**

```
T * idPtr = new T;
T * idPtr = new T[count];
```

où :

- T * : type de donnée pointé (celui qui sera réservé)
- idPtr : identifiant du pointeur
- `new` : opérateur d'allocation dynamique C++
- count : nombre d'éléments demandés de type T

Code source 29 – Allocation dynamique : new

```
1 // allocation dynamique d'un entier
2 int * idPtr = new int;
3 // allocation dynamique de 10 entiers
4 int * idPtr = new int[10];
```

L'utilisation du pointeur *idPtr* est ensuite identique à celui d'une valeur unique ou d'un tableau de 10 nombres.

**Attention**

```
1 int* ptr = new int(5);
```

alloue l'espace d'un seul entier initialisé à la valeur 5

### 1.10.2 Libération de l'espace mémoire

**1.10.2.1 Fonction `free` en C/C++** La fonction `free` permet de libérer l'espace mémoire alloué lorsqu'il n'est plus utile.

**Syntaxe 1.14 fonction free : prototype**

```
void free(void * ptr );
```

où :

- void : type retourné par la fonction (pas de retour ici)
- free : fonction de libération de l'espace mémoire
- ptr : pointeur vers l'espace alloué actuel

Code source 30 – Allocation dynamique : realloc

```
1 int * idPtr = (int *) malloc(10, sizeof(int));
2 ... // utilisation de l'espace
3 free(idPtr); // libération de l'espace mémoire
4 idPtr = NULL; // pour la propreté du code
```

L'instruction `free` peut être appelée sur un pointeur même si le pointeur vaut NULL : c'est, dans ce cas, une instruction nulle (en anglais : *nop, no operation*).

#### 1.10.2.2 Fonction `delete` en C++

La fonction `delete` permet de libérer l'espace mémoire alloué par `new` lorsqu'il n'est plus utile.

**Syntaxe 1.15 fonction delete : prototype**

```
void operator delete[] (void * ptr);
```

où :

- void : type retourné par la fonction (pas de retour ici)
- `delete` : fonction de libération de l'espace mémoire
- ptr : pointeur vers l'espace alloué actuel

Code source 31 – Allocation dynamique : realloc

```
1 int * idPtr1 = new int;
2 int * idPtr2 = new int[10];
3 ... // utilisation
4 delete(idPtr1); // libération de l'espace mémoire
5 delete[](idPtr2); // libération de l'espace mémoire
6 idPtr1 = NULL; // pour la propreté du code
7 idPtr2 = NULL; // pour la propreté du code
```

L'instruction `delete` peut être appelée sur un pointeur même si le pointeur vaut NULL : c'est, dans ce cas, une instruction nulle (en anglais : *nop, no operation*).

### 1.10.3 Allocation dynamique : retour d'une fonction

Attention : L'exemple suivant fonctionne "apparemment" :

Code source 32 – Fonction : type de retour pointeur

```
1  #include <stdio.h>
2  #define NB 12
3  int * ftab()
4  {
5          int tableau[NB];
6          for (int i=0; i < NB; ++i) {
7                  tableau[i] = 0;
8          }
9          return tableau;
10 }
11 int main()
12 {
13         int * tab;
14         tab = ftab();
15         for (int i=0; i < NB; ++i) {
```

```
16                  printf("␣%d␣", tab[i]);
17          }
18
19    return 0;
20  }
```

La compilation de cet exemple nous indique "warning : function returns address of local variable".

En effet, l'appel de la fonction fTab initie la déclaration d'un tableau à une certaine adresse mémoire, puis, lors qu'on quitte la fonction, on retourne une valeur, l'adresse de début du tableau, puis le tableau est desalloué automatiquement.

C'est la valeur de l'adresse du tableau qui est reçue dans l'instruction tab = ftab() mais l'espace pointé a été réutilisé (voir section 1.11 page 24).

#### 1.10.3.1 Allocation dynamique demandée par le programmeur

Lorsqu'une fonction doit retourner un tableau, il est nécessaire de passer par un pointeur et l'allocation dynamique.

L'exemple suivant illustre la méthode utilisée :

Code source 33 – Fonction : type de retour pointeur

```
1  #define NB 12
2  int * ftab()
3  {
4          int * ptr = malloc(NB * sizeof(int));
5          for (int i=0; i < NB; ++i) {
6                  ptr[i] = 0;
7          }
8          return ptr;
9  }
10 int main()
11 {
12         int * tab = ftab();
13         for(int i=0;i< NB;++i) {
14                 ptr[i] = i;
15         }
16         for(int i=0;i< NB;++i) {
17                 printf("␣%d",tab[i]);
18         }
19         free(tab);
20         tab = NULL;
21 }
```

Le mécanisme est identique avec les opérateurs `new` et `delete` de C++.

## 1.11 Allocation dynamique et mémoire vive

En réalité, l'espace mémoire dédié à l'allocation des données d'un programme est découpé en différentes zones :

— une zone statique des données, pour l'allocation de variables globales ou définies comme `static`, des constantes également, qui existeront tout au long de l'exécution ; c'est le principe d'allocation le plus sûr, mais il est rigide ;

— une zone dynamique, la *pile* (en anglais : *stack*), dédiée à l'allocation automatique de variables dans des blocs d'instructions : une fois le bloc quitté, les variables sont automatiquement desallouées [3] ; c'est un principe sûr car il utilise un empilement puis un dépilement au fur et à mesure de la libération ;

— et une zone dynamique, le *tas* (en anglais : *heap*), utilisé à la demande du programmeur : c'est dans cet espace que seront placées les variables allouées dynamiquement, à la demande ; c'est le principe le moins sûr, car à la charge du développeur et peut être source d'erreurs d'exécution.

## 1.12 Exercice

1. Soient trois entiers a, b et c et deux pointeurs de type entier ptr1 et ptr2. Effectuer la trace d'exécution du programme suivant :

Code source 34 – Exemple

```
1  int main()
2  {
3   int a = 0, b = 1, c = 2;
4   int * ptr1, *ptr2;
5
6   ptr1 = &a;
7   ptr2 = &c;
8
9   *ptr1 = *ptr2;
10  (*ptr2)++;
11  ptr1 = ptr2;
12  ptr2 = &b;
13  *ptr2 = *ptr1 -  2 * * ptr2;
14  (*ptr2)  --;
15  *ptr1 = *ptr2 - c;
16  a = (2 + *ptr2) * *ptr1;
17  ptr2 = &c;
18  *ptr2 = *ptr1 / *ptr2;
19  *ptr1 = a + b;
20  a += *ptr1;
21  b = *ptr1 + *ptr2;
22  *ptr1 = 2 * *ptr1;
23  a = *ptr2;
24  *ptr2 = *ptr1 -  *ptr2;
25  *ptr2 += *ptr1 + a;
26  ptr2 = ptr1 = &a;
27  ptr2++;
28  ptr1 += 2;
29  c = ptr2 == &c;
```

3. à moins d'avoir été qualifiées de `static`

```
30   ptr1 = NULL;
31
32   return 0;
33 }
```

2. Tester ce programme et représenter l'évolution de l'état mémoire.

Code source 35 – Exemple

```
1  #include <stdio.h>
2  void incrementer(int * v)
3  {
4   *v = ++(*v);
5  }
6  int main()
7  {
8   int a = 5;
9   int * ptr;
10
11  ptr = &a;
12
13  incrementer(&a);
14  printf("\na vaut maintenant %d ou %d ou %d", a, *ptr, *&a);
15  incrementer(ptr);
16  printf("\na vaut maintenant %d ou %d ou %d", a, *ptr, *&a);
17
18  return 0;
19 }
```

3. Nouveau programme :
   (a) Écrire la fonction `echanger`
      — elle attend 2 paramètres `x` et `y` de type entier
      — elle échange les contenus des variables `x` et `y` (permutation)
   (b) Écrire un programme qui demande à l'utilisateur la saisie de 2 entiers `a` et `b`, puis échange et affiche ces 2 entiers. On utilisera la fonction précédente. Exemple de déroulement :

```
Entrez l'entier a : 5
Entrez l'entier b : 12
a vaut 12
b vaut 5
```

   (c) Une véritable fonction de type `int` pourrait-elle être écrite pour remplacer `echanger` ?

Deuxième partie

# Algorithmes de tri

## 2 Tri

Le tri, ou classement, (en anglais : *sort*) est l'opération qui consiste à ordonner une suite d'objets selon un certain ordre de valeurs de clefs, un des membres de l'objet. (on pourra classer une liste d'étudiants par ordre numérique ou alphabétiquement)

On appliquera ici le tri sur des éléments simples comme un tableau de nombres.

### 2.1 Tri par insertion

Le principe du tri par sélection (en anglais : *insertion sort*) est le suivant : pour chaque élément du tableau du premier au dernier, rechercher son emplacement dans les éléments précédemment triés et l'insérer.

Au fur et à mesure de la progression du tri, la partie gauche du tableau est progressivement triée.

**Algorithme 1** Tri par insertion d'un tableau tab de taille N

```
1: i : entier ← 1
2: j : entier ← 0
3: tant que i < N faire
4:     j ← i
5:     tant que j > 0 et tab[j-1] > tab[j] faire
6:         Permutter tab[j] et tab[j-1]
7:         j ← j − 1
8:     fin tant que
9:     i ← i + 1
10: fin tant que
```

### 2.2 Tri par sélection

Le principe du tri par sélection (en anglais : *selection sort*) est le suivant : pour chaque élément du tableau du premier au dernier, rechercher si, parmi les suivants, il existe un élément dont la valeur est plus petite : si c'est le cas, permuter les 2 éléments.

Au fur et à mesure de la progression du tri, la partie gauche du tableau est progressivement triée.

**Algorithme 2** Tri par sélection d'un tableau tab de taille N

```
1: i : entier ← 0
2: j : entier ← 0
3: tant que i < N faire
4:     j ← i + 1
5:     tant que j < N faire
6:         si tab[i] > tab[j] alors
7:             Permutter tab[i] et tab[j]
8:         fin si
9:         j ← j + 1
10:    fin tant que
11:    i ← i + 1
12: fin tant que
```

## 2.3 Tri bulle

Le principe du tri bulle (en anglais : *bubble sort*) est le suivant : tant qu'il y aura eu des permutations, répéter pour chaque élément du tableau du premier au dernier, comparer l'élément par rapport au suivant : s'il est plus grand, les permuter.

Au fur et à mesure de la progression du tri, la partie droite du tableau est progressivement triée.

**Algorithme 3** Tri bulle d'un tableau tab de taille N

```
1: i : entier ← 0
2: perm : booléen
3: faire
4:     perm ← faux
5:     i ← 1
6:     tant que i < N faire
7:         si tab[i] < tab[i-1] alors
8:             Permutter tab[i] et tab[i - 1]
9:             perm ← vrai
10:        fin si
11:        i ← i + 1
12:    fin tant que
13: tant que perm
```

Troisième partie

# Structures complexes

## 3 Notion de Type de Donnée Abstrait

Les *types abstraits de données*, TAD, ((en anglais : *ADT, Abstract Data Type*))décrit de manière simplifiée une structure de donnée :

— les membres qui la composent

— les opérations qui lui sont associées

Quelques TAD bien formalisés :

— les piles

— les files

— les listes

— les graphes (les arbres en sont un cas particulier)

Lorsqu'on doit choisir un modèle de données s'appuyant sur ces structures, il est utile d'en comparer la complexité des opérations.

## 4 Pile, LIFO

**Pile**

La *pile* (en anglais : *stack*) permet l'empilement d'un certain nombre d'objets, en vue d'un dépilement ultérieur. Dans une pile, le dernier objet empilé sera le premier dépilé (dernier entré, premier sorti, (en anglais : *last in, first out, LIFO*). Voir figure 3 page 30)

Les opérations associée à la pile sont les suivantes :

— empiler un objet (en anglais : *push*), `empiler(obj)` : ajouter l'objet au sommet de la pile

— dépiler le dernier objet empilé (en anglais : *pop*), `depiler() : obj` : récupérer l'objet du sommet de la pile et l'enlever de la pile

— obtenir le nombre d'objets empilés, soit la taille actuelle de la pile

— déterminer si la pile est vide (auquel cas on ne pourra plus dépiler)

— déterminer si la pile est pleine (auquel cas on ne pourra plus empiler)

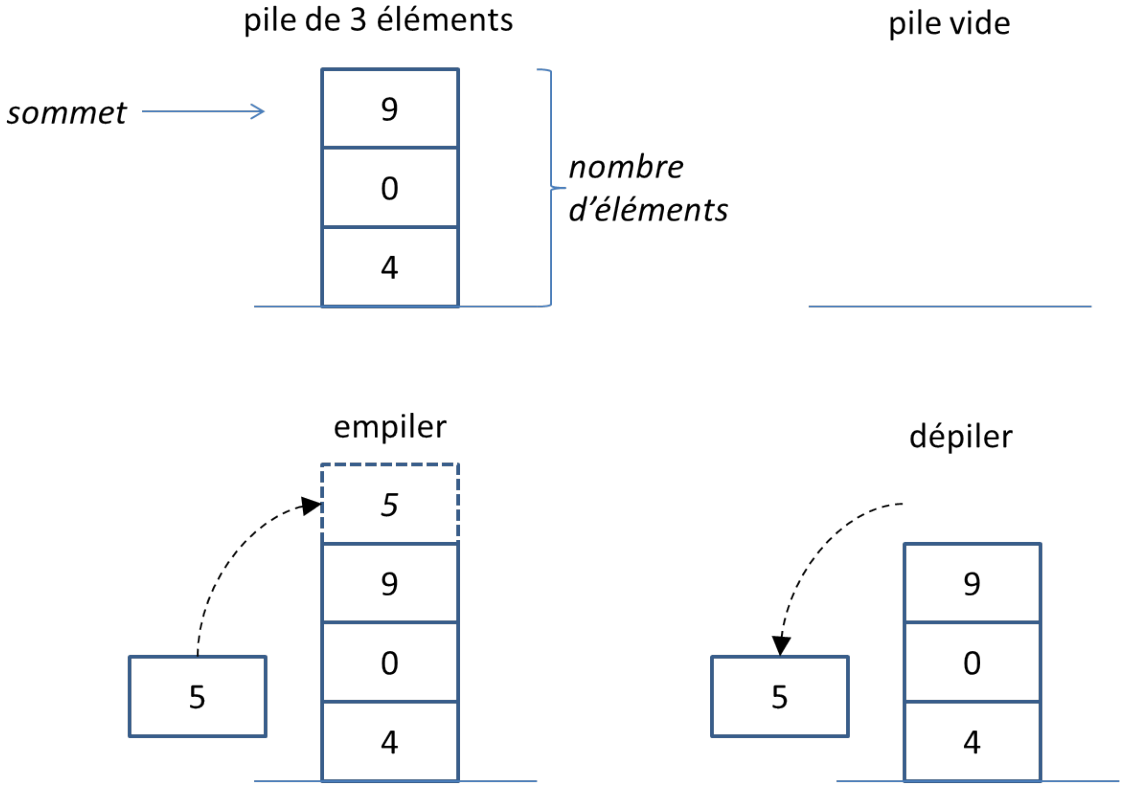
### 4.1 Pile de taille fixe

On peut simuler une pile de taille fixe en utilisant :

— un tableau d'objets

— un nombre entier représentant l'indice du sommet de la pile dans le tableau (on se mettra d'accord pour dire qu'un indice -1 représente une pile vide)

FIGURE 3 – Pile




```
enregistrement Pile
   liste [N] : T /* liste est un tableau de type T */
   sommet : entier
finEnregistrement
```

Les fonctions associées sont décrites ci-dessous.

**Algorithme 4** Initialiser une pile vide

1: **Fonction** INITPILE(()) : Pile
2: $p$ : Pile ▷ $p$ est une Pile
3: $p.sommet \leftarrow -1$
4: **return** $p$
5: **fin Fonction**

**Algorithme 5** Tester si une pile est pleine

1: **Fonction** ESTPILEPLEINE($p$ : Pile) : booleen
2: **return** $p.sommet >= N$
3: **fin Fonction**

**Algorithme 6** Empiler

```
1: Fonction EMPILER(p : Pile, o : objet)                    ▷ o, doit être empilé à p
Pre-condition: (!estPleine(p))
2:     p.sommet ← p.sommet + 1
3:     p.liste[p.sommet] ← o
4: fin Fonction
```

**Algorithme 7** Tester si une pile est vide

```
1: Fonction ESTPILEVIDE(p : Pile) : booleen
2:     return p.sommet < 0
3: fin Fonction
```

**Algorithme 8** Dépiler

```
1: Fonction DEPILER(p : Pile) : Objet
Pre-condition: (!estVide(p))
2:     o : Objet
3:     o ← p.liste[p.sommet]
4:     p.sommet ← p.sommet − 1
5:     return o
6: fin Fonction
```

**Algorithme 9** Obtenir la taille actuelle d'une pile

```
1: Fonction TAILLE(p : Pile) : entier
2:     return p.sommet + 1
3: fin Fonction
```

## 4.2 Pile dynamique

Le mécanisme de liste chainée permet de gérer une pile dynamique : l'empilement et le dépilement sont toujours réalisés en fin de la liste (voir section 6.2 page 35)

## 4.3 Exercice

1. A l'aide d'un tableau, créer une pile de nombres entiers (maximum de 128 nombres) ; le programme essaiera d'empiler 130 nombres aléatoires (entre 1 et 20) puis de dépiler 130 nombres.
2. un palindrome est un mot dont on peut inverser les lettres : utiliser une pile de lettres pour tester que chaque lettre d'un mot saisi est bien un palindrome (Aide : une fois empilées les lettres du mot, on pourra comparer la 1ere lettre au dépilement de la dernière, etc.)

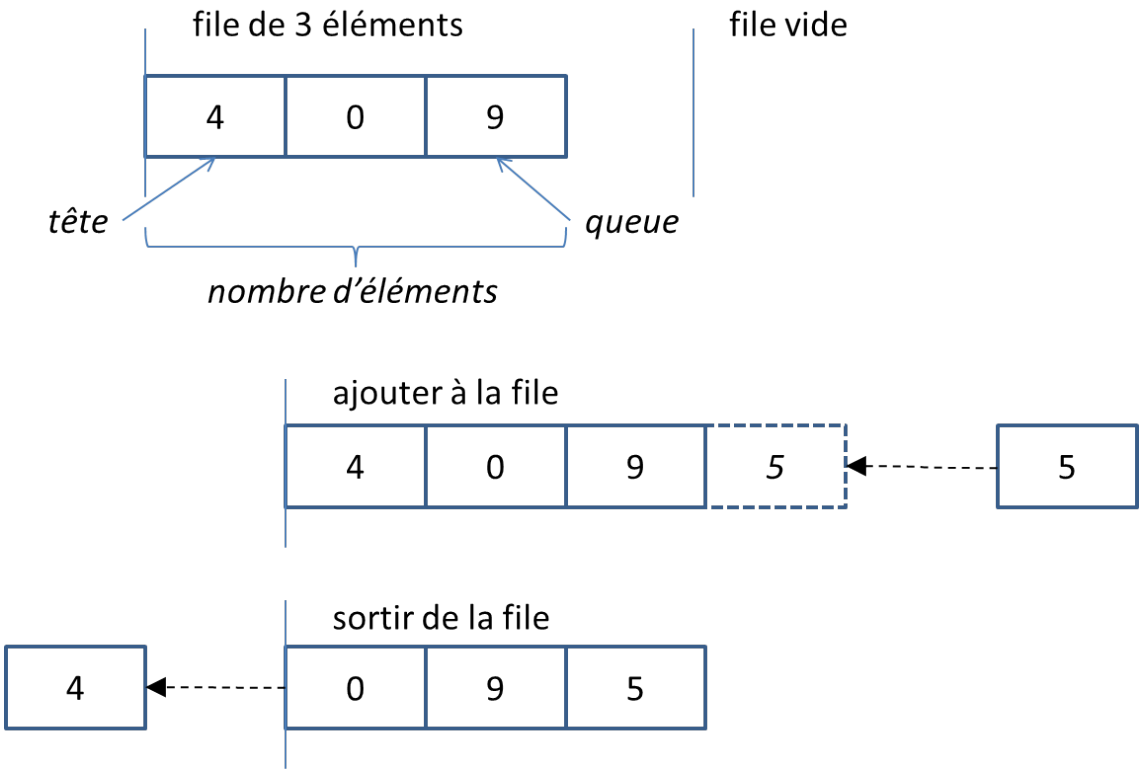
# 5 File, FIFO

**File**

La *file* (en anglais : *queue*) permet la gestion d'une suite ordonnée d'éléments à traiter (suite de travaux à réaliser dans un certain ordre, guichet bancaire ou caisse d'un magasin, etc.). Généralement, dans une file, la première valeur entrée sera la première sortie (premier entré, premier sorti, (en anglais : *first in, first out, FIFO*).

Voir figure 3 page 30

FIGURE 4 – File




Les principales opérations associée à la file sont les suivantes :

— ajouter à la file, enfiler (en anglais : *enqueue*), `enfiler(obj)` : ajouter un élément à la file

— sortir de la file, défiler (en anglais : *dequeue*), `defiler() : obj` : récupérer le premier élément de la file et le retirer

— obtenir le nombre d'objets enfilés, soit la taille actuelle de la file

— déterminer si la file est vide (auquel cas on ne pourra plus défiler)

— déterminer si la file est pleine (auquel cas on ne pourra plus enfiler)

## 5.1 File à taille fixe

On peut simuler une file à taille fixe en utilisant :

— un tableau d'objets

— un nombre entier représentant l'indice dans le tableau du sommet de la file (prochaine position d'entrée dans la file : on se mettra d'accord pour dire qu'un indice -1 représente une file vide)

```
enregistrement File
   liste [N] : T /* liste est un tableau de type T */
   sommet : entier
finEnregistrement
```

Les fonctions associées sont décrites ci-dessous.

**Algorithme 10** Initialiser une file vide

1: **Fonction** INITFILE(()) : file
2: $f$ : File ▷ $p$ est une file
3: $f.sommet \leftarrow -1$
4: **return** $f$
5: **fin Fonction**

**Algorithme 11** Tester si une file est pleine

1: **Fonction** ESTFILEPLEINE($f$ : File) : booleen
2: **return** $f.sommet >= N$
3: **fin Fonction**

**Algorithme 12** Emfiler

1: **Fonction** ENFILER($f$ : File, $o$ : objet) ▷ $o$, doit être enfilé à f
**Pre-condition:** (!$estPleine(f)$)
2: $f.sommet \leftarrow f.sommet + 1$
3: $f.liste[f.sommet] \leftarrow o$
4: **fin Fonction**

**Algorithme 13** Tester si une file est vide

1: **Fonction** ESTFILEVIDE($f$ : File) : booleen
2: **return** $f.sommet < 0$
3: **fin Fonction**

**Algorithme 14** Défiler

1: **Fonction** DEFILER($f$ : File) : Objet
**Pre-condition:** (!$estVide(p)$)
2: $o$ : Objet
3: $o \leftarrow f.liste[0]$
4: décaler les éléments à droite (indices 1 à f.sommet) d'une position vers la gauche
5: $f.sommet \leftarrow f.sommet - 1$
6: **return** $o$
7: **fin Fonction**

**Algorithme 15** Obtenir la taille actuelle d'une file

1: **Fonction** TAILLE($f$ : File) : entier
2: **return** $f.sommet + 1$
3: **fin Fonction**

## 5.2 File dynamique

Le mécanisme de liste chainée permet de gérer une file dynamique : l'ajout est toujours réalisé à la fin de la liste et la sortie est toujours réalisée en début de liste (voir section 6.2 page 35)

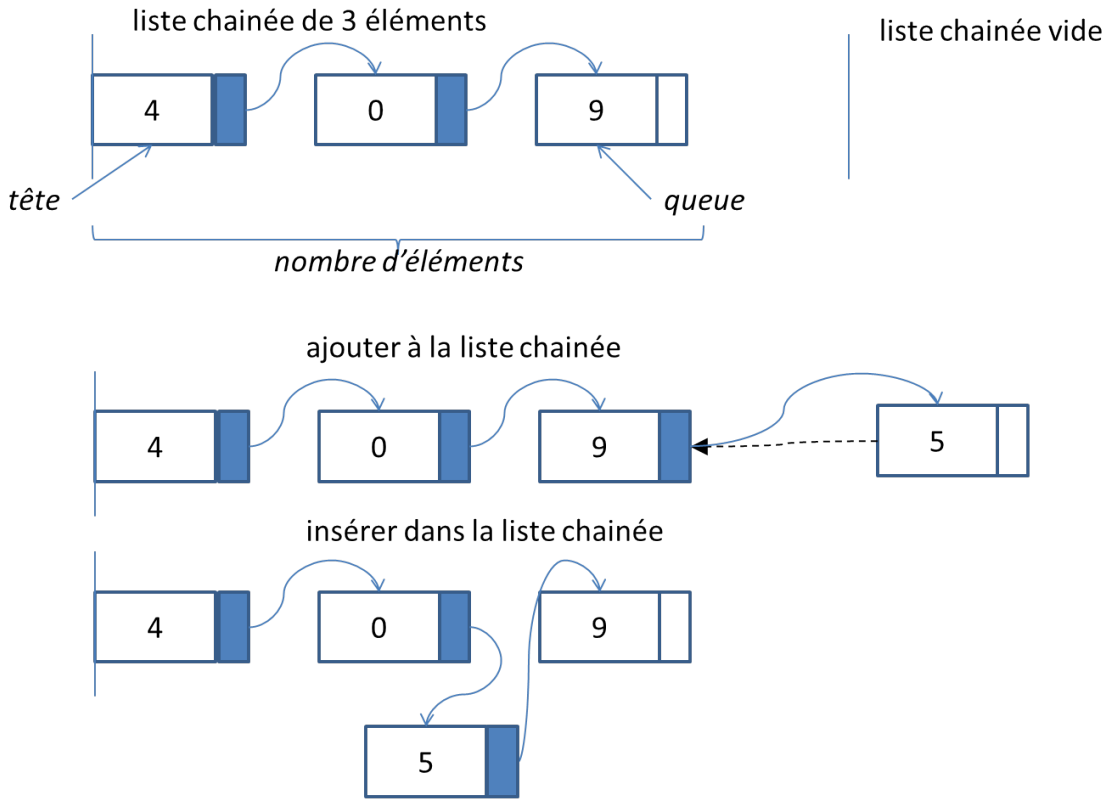
# 6 Listes chainées

**Liste chainée**

Une *liste chaînée* est une structure de données constituée d'éléments chaînés, où chaque élément possède des données propres et un pointeur vers l'élément suivant.

Voir figure 5 page 34

FIGURE 5 – Liste chainée




Le nombre d'élément de cette structure n'est pas limité (sauf par la taille mémoire disponible) : chaque élément peut être alloué dynamique puis être intégré à la liste chainée.

Les opérations associée à la liste chainée sont nombreuses, les principales sont les suivantes :

— ajouter à la fin : ajouter un élément à la fin de la liste

— ajouter en début : insérer un élément au début de la liste

— insérer : ajouter un élément à une certaine position de la liste, après ou avant un autre élément

— supprimer : supprimer un élément de la liste

— rechercher, parcourir : rechercher un élément de la liste (en vue d'une suppression ou d'une insertion)

## 6.1 Liste à taille fixe

On peut simuler une liste chainée à taille fixe en utilisant 1 tableau :

— un tableau de valeurs

— un nombre entier représentant l'indice dans le tableau du dernier élément de la liste (on s'accordera sur la value -1 comme indice de liste vide).

```
enregistrement Liste
   liste [N] : T /* liste est un tableau de type T */
   sommet : entier
finEnregistrement
```

Des vérifications devront être mises en oeuvre :

— on ne pourra ajouter un objet que si la liste n'est pas pleine

— on ne pourra sortir une valeur que si la liste n'est pas vide

— le maintien de la séquence des éléments devra être assuré à chaque insertion et suppression : décalage vers la droite ou vers la gauche des éléments.

## 6.2 Liste chainée dynamique

Pour gérer une liste chainée dynamique, on aura besoin :

— d'un enregistrement représentant un élément de la liste avec pour membres
  — les données portées par chaque élément
  — un pointeur vers l'élément qui lui succède (l'élément suivant)

— d'un enregistrement représentant la liste chainée elle-même et comportant
  — un pointeur vers le 1er élément de la liste (tête de la liste)
  — un pointeur vers le dernier élément de la liste (queue de la liste)

Code source 36 – Structure de donnée pour une liste chainée dynamique

```
1  struct Element {
2   T idVar;
3   Element * suivant;
4  };
5  typedef struct Element Element;
6
7  struct ListeChainee {
8   Element * tete, // tete de liste
9       * queue; // fin de liste
10 }
11 typedef struct ListeChainee ListeChainee;
```

où :

— *Element* : enregistrement définissant le contenu de chaque élément de la liste :
  — des données
  — un pointeur vers l'élément suivant (référence récursive vers le prochain élément de type *Element*) ; une valeur nulle du pointeur *suivant* indique qu'il n'y a pas d'élément suivant, l'élément courant est le dernier de la liste
— *ListeChainee*, comportant :
  — *tete* : pointeur vers le premier élément de la liste ; une valeur nulle du pointeur traduit une liste vide
  — *queue* : pointeur vers le dernier élément de la liste ; une valeur nulle du pointeur traduit une liste vide

**Algorithme 16** Parcourir une liste chainée : version itérative

```
1: Fonction PARCOURIRITER(ptrElem : Ptr)        ▷ ptrElem, pointeur vers un élément
2:     elem : Elem                                ▷ elem contenu d' un élément
3:     tant que ptrElem != NULL faire
4:         elem ← *ptrElem
5:         Ecrire elem
6:         ptrElem ← elem.suivant
7:     fin tant que
8: fin Fonction
```

**Parcours itératif** Si liste est une variable de type 'ListreChainee', on effectuera le parcours itératif par l'appel parcourirIter ( liste .premier);

**Parcours récursif** Les éléments d'une liste chainée étant une structure de donnée récursive, le parcours récursif peut être utilisé.

**Algorithme 17** Parcourir une liste chainée : version récursive

```
1: Fonction PARCOURIRREC(ptrElem : Ptr)         ▷ ptrElem, pointeur vers un élément
2:     elem : Elem                                ▷ elem contenu d' un élément
3:     si ((ptrElem = NULL)) alors
4:         return
5:     sinon
6:         elem ← *ptrElem
7:         Ecrire elem
8:         parcourirRec (elem.suivant)
9:     fin si
10: fin Fonction
```

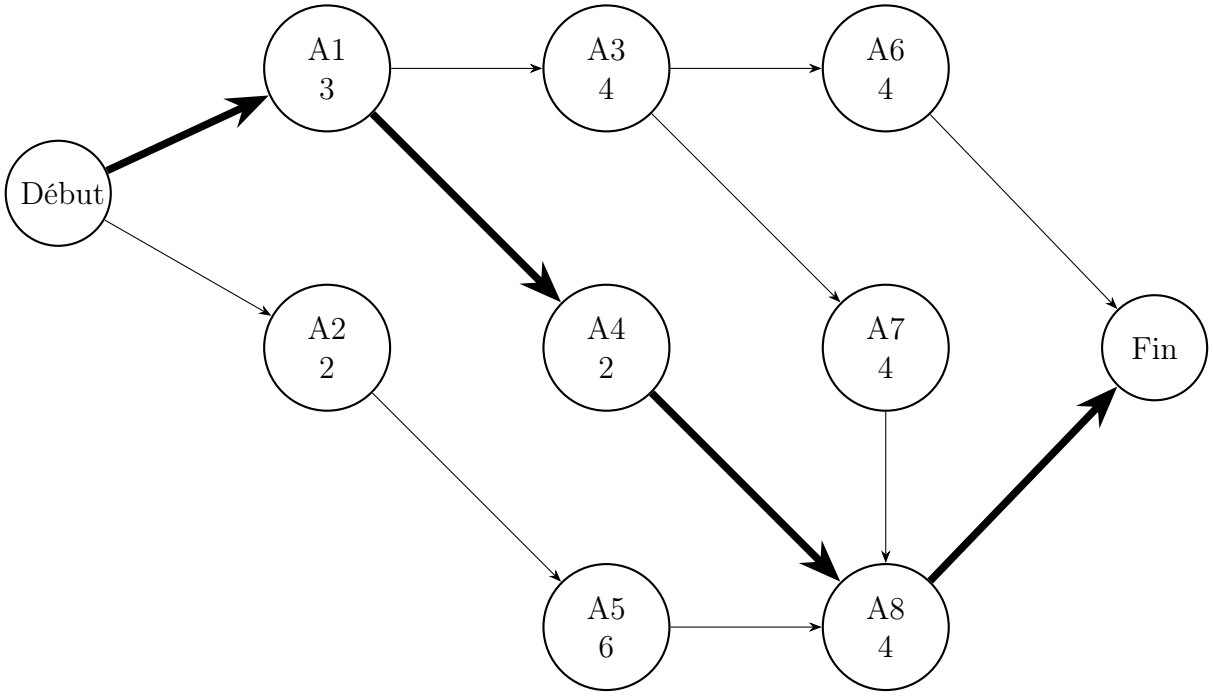
Si liste est une variable de type 'ListeChainee', on effectuera le parcours récursif par l'appel parcourirRec( liste .premier);

# 7 Graphes

Un graphe est un ensemble de points dont certains sont reliés 2 à 2. Les liaisons entre ces points peuvent être orientées ou non. Les points sont appelés sommets (en anglais : *vertice*) ou noeuds (en anglais : *nodes*), les liens sont appelés arêtes (en anglais : *edge*) ou arcs (orientés). Les graphes peuvent être étiquetés : aux sommets ou arêtes sont associés des valeurs d'un ensemble (nombres, couleurs, etc.)

Les graphes permettent la représentation de données complexes, non linéaires [4], comme : les hyperliens du Web, le réseau Internet, les réseaux sociaux, la succession des états d'un système, la planification (exemple ici d'un graphe PERT), etc.

FIGURE 6 – Graphe




Les arbres sont des cas particuliers de graphes : ils sont strictement hiérarchiques.

Les arbres binaires sont des cas particuliers d'arbres : un noeud ne peut être relié qu'à 2 noeuds fils.

## 7.1 Arbres binaires

Un *arbre binaire* (en anglais : *BT, Binary Tree*) est une structure de données de type graphe représentée sous forme hiérarchique, arborescente, dont chaque élément est appelé noeud (en anglais : *node*). Un noeud permet le stockage de données de manière efficace (pour la recherche), extensible (la limite est la mémoire) et cohérente (représentation de hiérarchies diverses, nomenclatures, familles, etc.).

Un noeud peut posséder au plus 2 noeuds fils : un noeud fils gauche et un noeud fils droit.

Le noeud initial d'un arbre binaire est appelé *racine*.

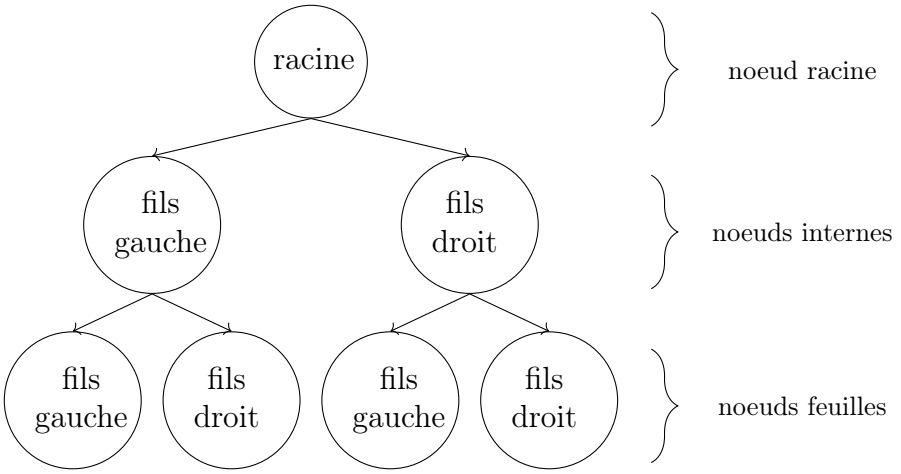
Les noeuds terminaux (ceux qui ne possèdent aucun noeud) sont appelés *feuilles*. Les autres noeuds sont appelés noeuds *internes*.

4. structures de données linéaires : tableaux ou listes chainées qui peuvent n'avoir qu'un successeur et qu'un prédécesseur

Les noeuds internes peuvent être considérés comme racines de sous-arbres. Le niveau d'un noeud dans un arbre est appelé profondeur.

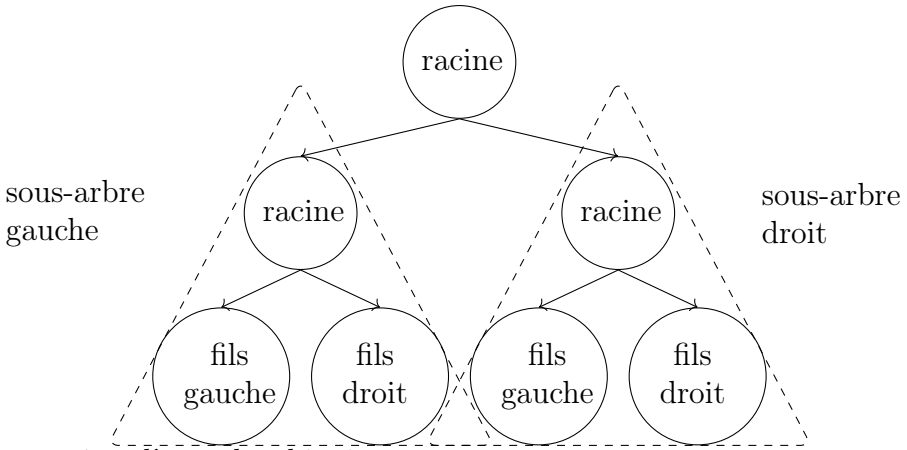
La *hauteur d'un arbre* binaire correspond à la distance entre la racine et la feuille la plus éloignée.

FIGURE 7 – Arbre



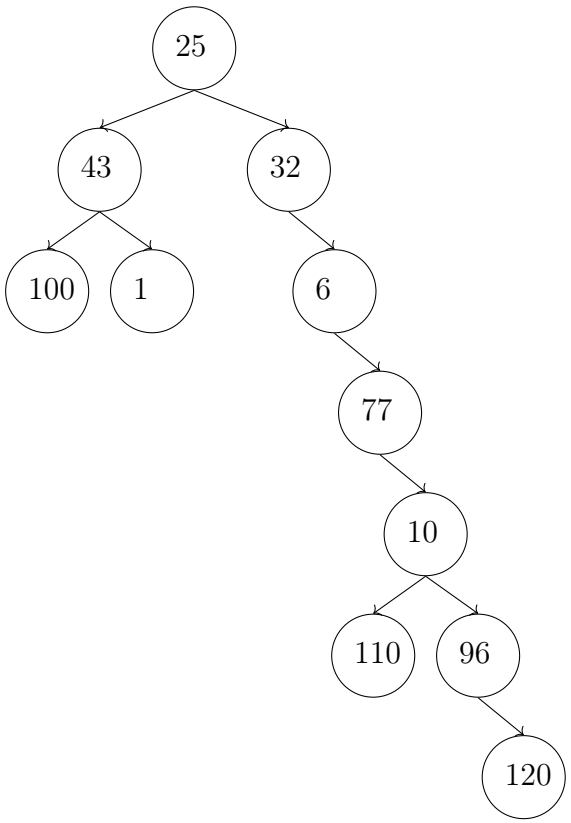

Chaque noeud peut être à son tour racine d'un sous-arbre :

FIGURE 8 – Arbres et sous-arbres




Exemple de représentation d'un arbre binaire

FIGURE 9 – Arbre binaire




### 7.1.1 Parcours en profondeur des arbres binaires

Des parcours dits "en profondeur" sont proposées pour le parcours des noeuds d'un arbre binaire (algorithmes récursifs) :

— le parcours *préfixe*

**Algorithme 18** parcours préfixe

```
1: Fonction parcourirPrefixe(Noeud n)
2:     afficher n
3:     si nonVide(gauche(n)) alors
4:         parcourirPrefixe (gauche(n))
5:     fin si
6:     si nonVide(droite(n)) alors
7:         parcourirPrefixe (droite(n))
8:     fin si
9: fin Fonction
```

affiche : 25, 43, 100, 1, 32, 6, 77, 10, 110, 96, 120

### 7.1.2 Représentation des arbres binaires en C

Code source 37 – Exemple de structure d'arbre binaire en C

```
1 struct noeud {
2   struct noeud * filsGauche;
```

```
3   struct noeud * filsDroit;
4   int valeur;
5 };
6 struct noeud * racine;
```

où :

- noeud est un enregistrement comportant 2 pointeurs vers ses fils (valeur nulle pour un noeud terminal) et des données (ici un nombre entier) ; la structure de donnée est récursive
- un pointeur vers le noeud racine ; une valeur nulle traduit un arbre vide.

Les opérations usuelles associées à l'arbre binaire sont les suivantes :

- ajouter un noeud : ajouter un noeud terminal en tant que noeud racine ou comme fils d'un autre noeud
- insérer un noeud : ajouter un noeud en modifiant l'arborescence
- supprimer un noeud : supprimer un noeud unique (terminal ou pas)
- supprimer une branche d'un noeud : supprimer un noeud et ses fils
- rechercher, parcourir : rechercher un noeuds dans l'arborescence

Quatrième partie

# Persistance des données

## 8 Persistance des données : fichiers

Les programmes sont amenés à gérer de nombreuses données qui sont demandées à l'utilisateur, calculées, puis perdues à la fin de l'exécution.

La persistance est la faculté à résister au temps qui passe (pensez aux arbres à feuillage persistant).

**Persistance des données**

Dans le domaine informatique, ce sont les mécanismes mis en oeuvre pour assurer une conservation durable des données d'un système d'information informatisé, lorsque le système informatique n'est plus alimenté en énergie. Au niveau d'un programme, il s'agit de mémoriser les données utilisées avant la fin de l'exécution et, éventuellement de les relire au démarrage de l'exécution.

Les système de persistance courants utilisent le fichier

— les fichiers gérés par le programme lui-même, par l'intermédiaire du système de gestion de fichier (SGF) [5] , ou système de fichiers (SF) (en anglais : *FS, File System*), du Système d'Exploitation ;

— les fichiers de base de données, gérés par un SGBD ; Système de Gestion de Base de Données (en anglais : *DBMS, DataBase Management System*).

Vis-à-vis du programme qui gère des données, les échanges d'informations sont considérés comme des flux (comme ceux des entrées-sorties clavier et écran).

Les flux (en anglais : *stream*) correspondent aux échanges réalisés entre un programme et les périphériques connectés à l'ordinateur : clavier, écran, système de gestion de fichiers, etc.

Les flux liés à la persistance des données ciblent les fichiers. Les fichiers permettent la conservation des informations saisies de manière permanente sur un support magnétique afin de pouvoir les réutiliser ultérieurement.

Comme la lecture du clavier et l'écriture vers l'écran, les flux de persistance vont également dans les 2 sens :

— système de fichier → programme : flux d'entrée (pour le programme),

— programme → système de fichier : flux de sortie (pour le programme).

L'utilisation d'un fichier (un flux pour le programme) comporte en général 3 phases :

— l'ouverture du fichier selon un certain mode (lecture, écriture, etc)

— l'accès aux données contenues dans le fichier (lecture), ou mémorisation de données dans le fichier (écriture)

— et la fermeture du fichier .

5. le système de gestion fichier correspond à un ensemble de mécanismes assurés par le Système d'Exploitation pour assurer l'organisation du stockage des données sur les supports mémoires de masse sous forme de fichiers et de répertoires

**Algorithme 19** Ecrire un fichier

```
1: f : fichier
2: Ouvrir f en mode Ecriture
3: tant que (condition) faire
4:     ...                          ▷ récupérer les valeurs à écrire dans le fichier (saisie)
5:     Ecrire dans f, listeDeValeurs
6: fin tant que
7: Fermer f
```

**Algorithme 20** Lire un fichier

```
1: f : fichier
2: Ouvrir f en mode Lecture
3: Lire f, listeDeValeurs                          ▷ 1ère lecture
4: tant que (!EOF(f)) faire          ▷ tant qu'on n'a pas atteint la fin du fichier f
5:     ...                           ▷ exploiter les valeurs lues à partir du fichier
6:     Lire f, listeDeValeurs
7: fin tant que
8: Fermer f
```

Le contenu d'un fichier peut être

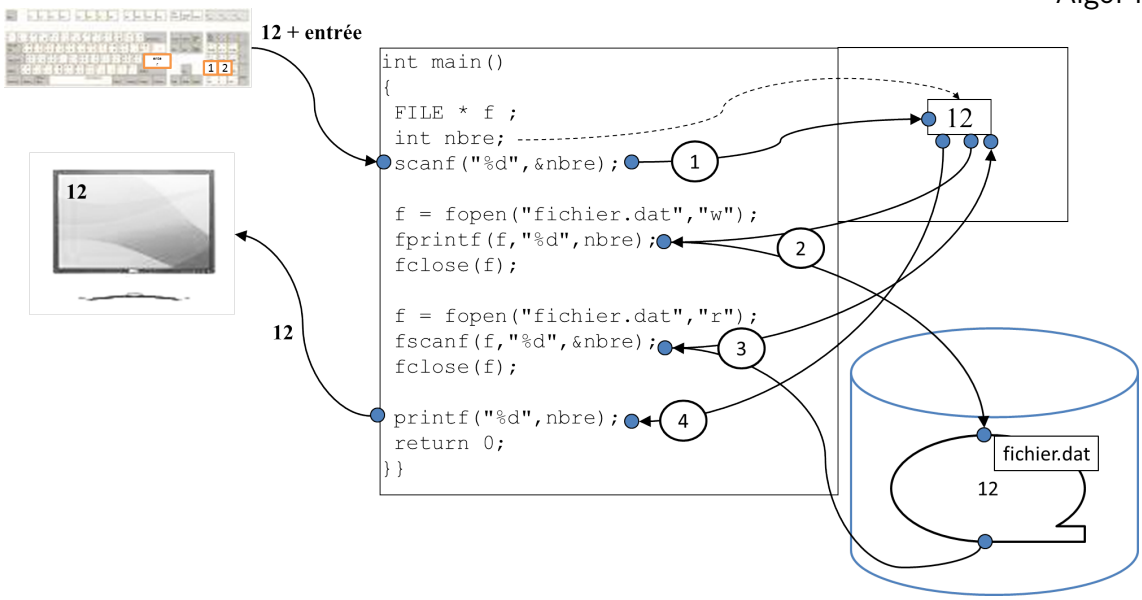
- organisé sous forme :
  - d'enregistrements au format unique mémorisés les uns à la suite des autres, sous forme de fichiers codés et texte brut (les données élémentaires sont séparées par un caractère défini) ou de fichiers comportant des données binaires (les données élémentaires sont stockées sous forme de blocs correspondant à des enregistrements)
  - de données balisées (format `xml`, par exemple)
- organisé sous la forme de texte comportant une grammaire : ce serait par exemple le code source d'un programme
- pas organisé : ce serait par exemple le contenu d'un mail

L'accès au contenu du fichier peut être :

- séquentiel : on accède aux informations les unes après les autres
- direct : on accès à une information particulière directement ; cela nécessite la connaissance de sa position dans le fichier ; un système de calcul de clef
- indexé : une clef est définie pour le fichier et on peut accéder directement à un enregistrement à partir de cette dernière

La bibliothèque `stdio` (cstdio en C++) met à disposition du programmeur un ensemble de fonctions permettant la gestion des entrées/sorties vers les fichiers (inclure le fichier d'en-tête <stdio.h> ou <cstdio>)

FIGURE 10 – Entrées/sorties fichiers




## 8.1 Fichiers textes brut

Les fichiers textes sont constitués uniquement de caractères ASCII visibles (mis à part les caractères de fin de ligne : CR LF (Carriage Return, Line Feed), OD OA en hexadécimal.

La longueur de chaque ligne dépend de la taille des valeurs qui y ont été mémorisées. Chaque valeur est séparée de la suivante par un caractère séparateur (espace, point-virgule) qui va permettre la relecture.

Un fichier texte peut être ouvert par un éditeur de texte : mais attention aux modifications qui pourraient nuire à la relecture du fichier par le programme.

### 8.1.1 Déclaration d'un flux

La déclaration d'un flux correspond à la réservation d'un espace mémoire permettant les échanges avec le SGF.

**Syntaxe 8.16 Déclaration d'un pointeur vers fichier en C/C++**

```
FILE * idPtr;
```

où :

- `FILE` : type correspond à une flux fichier
- `FILE *` : pointeur vers le type fichier
- *idPtr* : identifiant du pointeur

Code source 38 – Déclaration d'un pointeur vers fichier

```
13  FILE * fn = NULL;
14  char * fname = "agenda.txt";
15  char prenom [24],
16      teleph[24];
17  int ct = 0;
```

**En C++ :** les entêtes `iostream` et `fstream` donnent accès aux types (objets) `ofstream` (en sortie) et `ifstream` (en entrée) :

**Syntaxe 8.17 Déclaration d'un fichier en C++**

```
ofstream f1 ;
ifstream f2 ;
```

### 8.1.2 Ouverture d'un flux

L'ouverture d'un flux effectue d'une demande d'accès au SGF selon un certain mode et retourne un pointeur vers fichier :

**Syntaxe 8.18 Ouverture d'un fichier : `fopen`**

```
FILE *fopen( const char *filename, const char *mode ) ;
```

où :

- `FILE *` : pointeur vers le type fichier
- `fopen` : nom de la fonction d'ouverture de fichier
- *filename* : nom du fichier sous forme d'une chaine de caractères
- *mode* : mode d'accès au fichier sous forme d'une chaine de caractères

TABLE 1 – Principaux modes d'accès aux fichiers en C

| mode | signification | le fichier existe | le fichier n'existe |
|---|---|---|---|
| "r" | read, ouvre un fichier en lecture | lecture à partir du début | erreur |
| "w" | write, ouvre un fichier en écriture | efface le contenu existant | crée un fichier vide |
| "a" | append, ouvre un fichier pour le compléter | préserve le contenu existant | crée un fichier vide |

En cas de succès, la fonction retourne un pointeur vers le fichier.

En cas d'erreur, la fonction retourne la valeur NULL.

Code source 39 – Ouverture d'un fichier en complément fichier

```
20  fn = fopen(fname, "a");
```

**Attention**

L'ouverture d'un flux de sortie créée le fichier s'il n'existe pas, mais peut l'écraser s'il existe déjà (mode "w").

**En C++ :** :

**Syntaxe 8.19 Déclaration d'un fichier en C++**

```
ofstream f1;
ifstream f2;

f1.open("fichier1.txt", ios::out); // ou ios::app
f2.open("fichier2.txt", ios::in);
if (f1.is_open()) ...
```

### 8.1.3 Écriture d'un fichier : écriture formatée avec `fprintf`

**Syntaxe 8.20 Ecrire dans un fichier texte : `fprintf`**

```
int fprintf( FILE *stream, const char *format, ... );
```

où :

- `int` : type de retour de la fonction
- `fprintf` : nom de la fonction d'écriture de fichier
- `FILE *` *stream* : pointeur vers le fichier
- `const char` * format : chaine de caractères des codes de format des données écrites
- ... : liste des valeurs des données sources, celle à écrire

Code source 40 – Ecriture d'un fichier

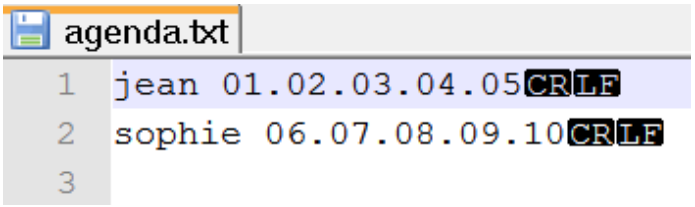
```
22      /* mémoriser les données :
23         demander la saisie d'un prénom
24         tant que le prenom ne commence pas par "*",
25              demander le téléphone et les enregistrer,
26              puis demander un nouveau prénom */
27      printf("\nSaisir le prenom : ");
28      scanf("%s",prenom);
29      while (prenom[0] != '*') {
30        scanf("%s",teleph);
31        fprintf(fn,"%s %s\n", prenom, teleph);
32        printf("\nautre prenom : ");
33        scanf("%s",prenom);
34      }
```

Le résultat de l'exécution est le suivant :

```
Saisir le prenom : jean
01.02.03.04.05

autre prenom : sophie
06.07.08.09.10
```

`autre prenom : *`

Le contenu du fichier :




**En C++ :**

**Syntaxe 8.21 Ecriture d'un fichier en C++**

```
f1 « "litteral" « donnee ;
```

### 8.1.4 Lecture d'un fichier : lecture formatée avec `fscanf`, détection de fin de fichier `feof`

**Syntaxe 8.22 Lire un fichier texte : `fscanf`**

```
int fscanf( FILE * stream, const char *format, ... ) ;
```

où :

- `int` : type de retour de la fonction
- `fscanf` : nom de la fonction de lecture de fichier
- `FILE *` stream : pointeur vers le fichier
- `const char` *format : chaine de caractères des codes de format des données lues
- ... : liste des adresses des variables réceptrices

La fonction `feof` permet la détection de la fin d'un fichier :

**Syntaxe 8.23 Tester la fin d'un fichier texte : `feof`**

```
int feof( FILE *stream ) ;
```

Elle retourne la valeur différente de 0 (assimilable au booléen `true`) si la fin du fichier est détectée, 0 sinon (assimilable au booléen `false`).

Code source 41 – Lecture d'un fichier

```
38    /* ouverture du fichier en lecture */
39  fn = fopen(fname,"r");
40    /* lecture des données
41       lire un prénom, tant qu'on n'est pas à la fin du fichier,
42         lire le numéro de téléphone
```

```
43          afficher les informations et lire le prénom suivant */
44    ct = 0;
45    fscanf(fn,"%s",prenom);
46    while (!feof(fn)) {
47      fscanf(fn,"%s",teleph);
48      ++ct;
49      printf("\n%d : %s %s", ct, prenom, teleph);
50      fscanf(fn,"%s",prenom);
51    }
52      /* fermer le fichier */
53    fclose(fn);
54    fn = NULL;
```

```
Saisir le prenom : jean
01.02.03.04.05

autre prenom : sophie
06.07.08.09.10

autre prenom : *

1 : jean 01.02.03.04.05
2 : sophie 06.07.08.09.10
```

**En C++ :**

**Syntaxe 8.24 Lecture d'un fichier en C++**

f2 » variable ;

### 8.1.5 Lecture et écriture par lignes complètes : `fputs` et `fgets`

Ces fonctions permettent l'écriture de lignes complètes sans s'occuper d'un format de donnée.

**Syntaxe 8.25 Ecrire et lire un fichier texte par blocs : `fputs`, `fgets`**

int fputs( const char *str, FILE *stream ) ; char *fgets( char *str, int count, FILE *stream ) ;

où :

- `int` : type de retour de la fonction
- `char *` : type de retour de la fonction
- `fputs` : nom de la fonction d'écriture de fichier

— `fgets` : nom de la fonction de lecture de fichier

— *count* : nombre de caractères lus (taille de str) ; count-1 caractères sont lus afin d'ajouter la caractère de fin de chaine

— `FILE *` stream : pointeur vers le fichier

— `const char` *str, char *str : chaine de caractères à écrire

Code source 42 – Lecture d'un fichier

```
13          const int LONGUEUR = 200;
14    FILE * fn = NULL;
15    char * fname = "agenda2.txt";
16    char ligne[LONGUEUR];
17    int ct = 0;
18          /* INITIALISATION */
19      /* ouverture du fichier en écriture */
20    fn = fopen(fname,"a");
21          /* TRAITEMENT */
22      /* mémoriser les données :
23        demander la saisie d'une ligne
24        tant que la ligne ne commence pas par "*",
25            enregistrer la ligne,
26            puis demander une nouvelle ligne */
27      printf("\nSaisir un texte :\n");
28      fgets(ligne,LONGUEUR,stdin);
29      /* traitement */
30      while (ligne[0] != '*') {
31        fputs(ligne, fn);
32        fgets(ligne,LONGUEUR,stdin);
33      }
34      /* fermer le fichier */
35    fclose(fn);
36    fn = NULL;
37      /* ouverture du fichier en lecture */
38    fn = fopen(fname,"r");
39      /* lecture des données
40        lire une ligne,
41        tant qu'on n'est pas à la fin du fichier,
42          afficher la ligne et lire la ligne suivante */
43    ct = 0;
44    fgets(ligne,LONGUEUR,fn);
45    while (!feof(fn)) {
46      ++ct;
47      printf("%d %s", ct, ligne);
48      fgets(ligne,LONGUEUR,fn);
49    }
50      /* fermer le fichier */
51    fclose(fn);
52    fn = NULL;
```

Résultat de l'exécution :

```
Saisir un texte :
pierre 01.02.03.04.05 anniv. le 3 avril toujours en retard !
sophie 06.07.08.09.10 anniv. le 5 ou 6 mai
*
1 pierre 01.02.03.04.05 anniv. le 3 avril toujours en retard !
2 sophie 06.07.08.09.10 anniv. le 5 ou 6 mai
```

Le contenu du fichier :

agenda2.txt

```
1 pierre 01.02.03.04.05 anniv. le 3 avril toujours en retard !CRLF
2 sophie 06.07.08.09.10 anniv. le 5 ou 6 maiCRLF
3
```

**En C++ :**

**Syntaxe 8.26 Ecriture d'un fichier en C++**

```
string ligne;
getline (f2,ligne)
```

### 8.1.6 Fermeture d'un fichier

La fermeture d'un flux enregistre les dernières données et libère la zone mémoire allouée pour accéder au fichier.

**Syntaxe 8.27 Fermeture d'un fichier : `fclose`**

```
int fclose( FILE *stream );
```

où :

- `int` : type de retour de la fonction
- `fclose` : fonction de fermeture de fichier
- `FILE *` stream : pointeur vers le flux

La fonction retourne 0 en cas de succès, et la valeur `EOF` en cas d'erreur. Le fichier n'est pus accessible après l'appel de la fonction.

Code source 43 – Fermeture d'un fichier

```
51  }
52    /* fermer le fichier */
```

**En C++ :**

**Syntaxe 8.28 Déclaration d'un fichier en C++**

```
f1.close();
f2.close();
```

## 8.2 Fichiers binaires

Alors que les fichiers textes sont constitués uniquement de caractères ASCII visibles, les fichiers binaires sont codés en fonction du type de donnée : les chaînes de caractères seront codées en ASCII, les nombres entiers en binaire, les nombres réels en binaire au format IEEE-754.

Contrairement aux fichiers textes, leur contenu n'est généralement pas lisible directement.

Les fichiers binaires sont adaptés au stockage des données structurées. Mais la longueur du type doit être fixe et connue à l'avance.

Un enregistrement correspond aux données relatives à la structure gérée (un bloc de données de longueur fixe). Un fichier binaire comporte autant d'enregistrements que de structures stockées. Chaque enregistrement peut être accédé directement à partir de sa position dans le fichier.

### 8.2.1 Ouverture d'un flux binaire

La fonction `fopen` est identique au fichier texte, c'est le mode d'ouverture qui change :

— "rb" : pour lecture binaire

— "ab" : pour complément binaire

— "wb" : pour écriture binaire

### 8.2.2 Ecriture et lecture directe : fonctions `fwrite` et `fread`

Ces fonctions écrivent et lisent des blocs organisés sous forme d'enregistrement .

**Syntaxe 8.29 Ecrire et lire un fichier binaire : `fwrite`, `fread`**

```
size_t fwrite( const void *buffer, size_t size, size_t count, FILE *stream );
size_t fread( void * buffer, size_t size, size_t count, FILE *stream )
```

Code source 44 – Ecriture et lecture d'un fichier binaire

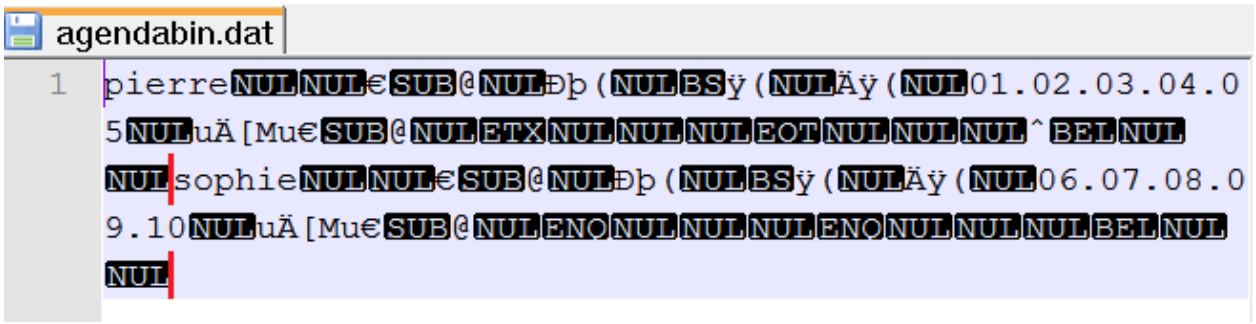
```
24  FILE * fn = NULL;
25  char * fname = "agendabin.dat";
26  Agenda ag;
27  int ct = 0;
28        /* INITIALISATION */
29    /* ouverture du fichier en écriture */
30  fn = fopen(fname,"ab");
31        /* TRAITEMENT */
```

```
32      /* mémoriser les données :
33        demander la saisie d'un prénom
34        tant que le prenom ne commence pas par "*",
35            demander le téléphone et les enregistrer,
36            puis demander un nouveau prénom */
37      printf("\nSaisir le prenom : ");
38      scanf("%s",&ag.prenom);
39      while (ag.prenom[0] != '*') {
40        scanf("%s",&ag.teleph);
41        scanf("%d %d %d", &ag.nais.jj, &ag.nais.mm, &ag.nais.aaaa );
42        fwrite(&ag, sizeof(ag), 1, fn);
43        printf("\nautre prenom : ");
44        scanf("%s",&ag.prenom);
45      }
46      /* fermer le fichier */
47    fclose(fn);
48    fn = NULL;
49      /* ouverture du fichier en lecture */
50    fn = fopen(fname,"rb");
51      /* lecture des données
52        lire un enregistrement, tant qu'on n'est pas à la fin du fichier,
53          afficher les informations et lire l'enregistrement suivant */
54    ct = 0;
55    fread(&ag, sizeof(ag),1,fn);
56    while (!feof(fn)) {
57      ++ct;
58      printf("\n%d : %s %s nais. %d/%d/%d",
59          ct, ag.prenom, ag.teleph, ag.nais.jj, ag.nais.mm, ag.nais.aaaa);
60      fread(&ag, sizeof(ag),1,fn);
61    }
62      /* fermer le fichier */
63    fclose(fn);
64    fn = NULL;
```

```
Saisir le prenom : pierre
01.02.03.04.05
3 4 1928

autre prenom : sophie
06.07.08.09.10
5 5 1933

autre prenom : *

1 : pierre 01.02.03.04.05 nais. 3/4/1928
2 : sophie 06.07.08.09.10 nais. 5/5/1933
```

Le contenu du fichier n'est maintenant plus directement lisible par un éditeur de texte car les valeurs numériques ne sont pas décodées :

FIGURE 11 – Contenu fichier 'agendabin'




### 8.2.3 Accès direct à une position de fichier : fonction `fseek`

La fonction `fseek` positionne le curseur de lecture à un certain endroit du fichier

**Syntaxe 8.30 Acces directe dans un fichier binaire : `fseek`**

```
int fseek( FILE *stream, long offset, int origin );
```

où :

- `FILE *` stream : pointeur vers le fichier est la variable fichier
- *offset* : déplacement en nombre d'octets à partir de l'origine spécifié au paramètre qui suit
- *origin* : parmi
  - `SEEK_SET` : début de fichier,
  - `SEEK_CUR` : position courante,
  - `SEEK_END` : fin du fichier.

Code source 45 – Lecture d'un fichier

```
28      /* ouverture du fichier en écriture */
29    fn = fopen(fname,"rb");
30          /* TRAITEMENT */
31      /* acceder directement au dernier enregistrement */
32    if (fseek(fn,-1*sizeof(ag),SEEK_END) == 0) {
33      fread(&ag, sizeof(ag),1,fn);
34      printf("\nlu : %s %s nais. %d/%d/%d",
35          ag.prenom, ag.teleph, ag.nais.jj, ag.nais.mm, ag.nais.aaaa);
36    }
37    else {
38      printf("enregistrement inaccessible (fichier vide ?)");
39    }
40      /* fermer le fichier */
41    fclose(fn);
42    fn = NULL;
```

```
lu : sophie 06.07.08.09.10 nais. 5/5/1933
```

## 8.3 Gérer les erreurs d'accès aux fichiers

Les entrées-sorties dépendent d'éléments extérieurs : ceux-ci ne sont pas forcément ceux attendus, ils peuvent être inexistants ou indisponibles, et l'appel aux fonctions peut échouer. Il est alors indispensable de savoir si une opération s'est bien déroulée ou pas.

Le premier point de contrôle se situe à l'ouverture du fichier. Après l'ouverture, le pointeur fichier a dû être affecté d'une valeur non nulle . Si cette valeur est nulle, cela signifie que ce fichier n'a pas été ouvert correctement.

Code source 46 – capture d'erreur à l'ouverture d'un fichier

```
1 fn = fopen(fname,"r");
2 if (fn == NULL) {
3   printf("erreur d'ouverture") ;
4   exit(EXIT_FAILURE) ; // cf. remarque plus bas
5 }
```

Les fonctions retournent généralement une valeur qui permet de savoir si celle-ci s'est bien déroulée ou pas.

Par exemple pour la fonction `fscanf` (identique pour les fonctions scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf) : ces fonctions retournent le nombre d'éléments lus de manière correcte et affectés (celui-ci pouvant être inférieur à celui prévu).

Code source 47 – capture d'erreur sur fscanf/scanf

```
1   int res
2   res = fscanf(f,"%s %s",nom,prenom);
3   /* si 2 valeurs non pas été lues, erreur */
4   if (res != 2) {
5     printf("pb lecture : %d %d",errno, res);
6     exit(EXIT_FAILURE);
7   }
```

Une valeur `EOF` (constante fournie) est renvoyée si la fin de l'entrée est atteinte avant que la 1ère conversion ait pu avoir lieu ou si une correspondance échoue. `EOF` est aussi renvoyé en cas d'erreur de lecture auquel cas une valeur d'erreur est affectée à `errno`.

## 8.4 Accès aux bases de données

La persistance des données est majoritairement assurée par les SGBD.

Des bibliothèques (pilotes d'accès) fournies par les différents éditeurs permettent d'envoyer des requêtes SQL de gestion des données d'une base de données.

L'exemple suivant montre la logique d'accès aux données :

Code source 48 – Lecture d'un fichier

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <winsock.h>
4 #include "MYSQL/mysql.h"
5
6 /*
```

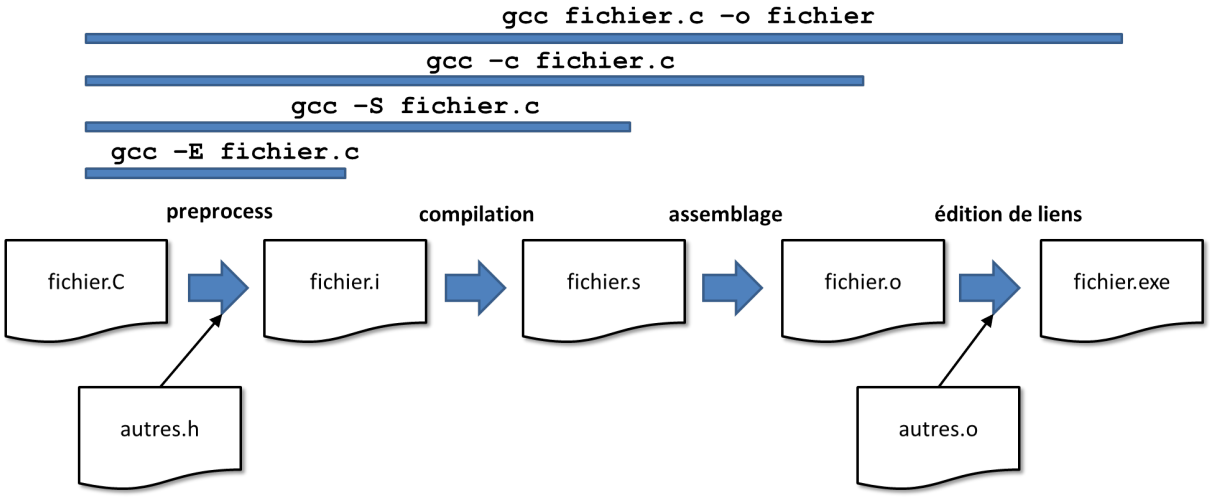
```
 7  * @but : accès aux bases de données avec SQL (MySQL ici)
 8  * @auteur : moi
 9  * @date : 2017/09/01
10  */
11
12 int main(void)
13 {
14         /* DECLARATIONS, INITIALISATIONS */
15     /* Déclaration d'un enregistrement de type MYSQL */
16   MYSQL conn;
17     /* Initialisation des données de la variable précédente */
18    mysql_init(&conn);
19     /* Options de connexion */
20    mysql_options(&conn,MYSQL_READ_DEFAULT_GROUP,"option");
21
22         /* TRAITEMENT */
23     /* si la connection au SGBD s'est bien passée */
24    if(mysql_real_connect(&conn,"localhost","root","password","tp1",0,NULL,0))
25       /* soumission de la requête au SGBD */
26     mysql_query(&conn, "SELECT num_act,nom_act FROM acteur order by nom_act"
27
28     /* Déclaration des variables résultat */
29     MYSQL_RES *result = NULL;
30     MYSQL_ROW row;
31
32     unsigned int i = 0;
33     unsigned int num_champs = 0;
34
35     /* affecter à result le jeu de données renvoyé par le SGBD */
36     result = mysql_use_result(&conn);
37
38     /* récupérer le nombre de colonnes du jeu de résultat */
39     num_champs = mysql_num_fields(result);
40
41     /* tant qu'il y a des lignes dans le résultat
42       affecter à la variable row la prochaine ligne du jeu de résultat */
43     while ((row = mysql_fetch_row(result)))
44     {
45       /* déclare un pointeur long non signé pour y stocker la taille des val
46       unsigned long *lengths;
47
48       /*On stocke ces tailles dans le pointeur */
49       lengths = mysql_fetch_lengths(result);
50
51       /* On fait une boucle pour accéder à la valeur de chaque colonne */
52       for(i = 0; i < num_champs; i++) {
53         // affichage d'un champ
54         printf("[%.*s] ", (int) lengths[i], row[i] ? row[i] : "NULL");
55       }
```

```
56      }
57
58      /* libération du jeu de résultat */
59      mysql_free_result(result);
60
61
62      /* libération de la connection au SGBD */
63      mysql_close(NULL);
64        //mysql_close(&conn);
65    } else {
66      printf("Une erreur s'est produite lors de la connexion au SGBD");
67    }
68
69    return 0;
70
71 }
```

# Cinquième partie

# Annexes

## 9 Chaine de compilation




Lorsque la compilation requiÃ¨re plusieurs fichiers, chacun est compilÃ© sÃ©parÃ©ment, puis l'ensemble est rassemblÃ© en un seul exÃ©cutable :

— compilation des diffÃ©rents fichiers sources

Code source 49 – Commande de compilation (gcc)

```
1 gcc -std=c99 -Wall -Wextra -Wpedantic -c source1.c
2 gcc -std=c99 -Wall -Wextra -Wpedantic -c source2.c
3 gcc -std=c99 -Wall -Wextra -Wpedantic -c sourceMain.c
```

— production du fichier exÃ©cutable

Code source 50 – Commande de compilation (gcc)

```
4 gcc -o source.exe source1.o source2.o sourceMain.o
```

.