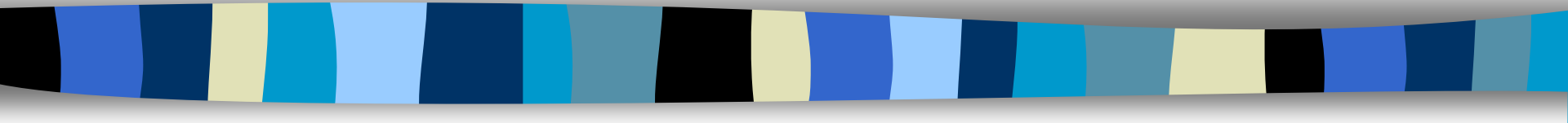


# chapitre 4- Utilisation de Piles et de Files



Ressource R1.01 : Initiation au développement - Partie 2

# Plan

---

<u>Définitions préalables</u>	3
<b><u>Piles</u></b>	8
– <u>Définition</u>	9
– <u>Caractéristiques</u>	11
– <u>Opérations sur une Pile</u>	12
• <u>Notation</u>	12
• <u>Initialiser</u>	13
• <u>Observateurs</u>	15
• <u>Primitives modifiant la pile</u>	23
• <u>Synthèse</u>	29
– <u>Erreurs liées à l'utilisation d'une pile</u>	30
– <u>Contenu du module de gestion de piles</u>	33
– <u>Exemple d'utilisation</u>	35
<b><u>Files</u></b>	40
– <u>Définition</u>	41
– <u>Caractéristiques</u>	44
– <u>Opérations sur une File</u>	45
• <u>Notation</u>	45
• <u>Initialiser</u>	46
• <u>Observateurs</u>	48
• <u>Primitives modifiant la file</u>	56
• <u>Synthèse</u>	62
– <u>Erreurs liées à l'utilisation d'une file</u>	63
– <u>Contenu du module de gestion de files</u>	66
– <u>Exemple d'utilisation</u>	68

---

# 0.- Définitions préalables (1/5)

---

## □ Introduction

- Les structures de données qui seront étudiées dans ce chapitre sont unes des plus connues parmi une infinité de structures possibles.
- Il s'agit
  - des piles
  - des files
- Elles sont très fréquemment rencontrées, tant dans la vie de tous les jours que dans les applications informatiques

# 0.- Définitions préalables (2/5)

---

## □ Structure de données **homogène**

### – Définition

C'est une structure de données regroupant un ensemble d'éléments de même type

### – Exemples

- Un tableau de caractères
- Un fichier de produits
- Une pile d'assiettes
- Une file de personnes

# 0.- Définitions préalables (3/5)

---

- Structure de données homogène **générique**
  - Définition  
C'est une structure de données homogène regroupant un ensemble d'éléments de même type, mais *a priori* non défini
  - Contre-exemple
    - Les *chaînes de caractères* ne sont pas génériques, car les éléments sont tous d'un même type défini : caractère

# 0.- Définitions préalables (4/5)

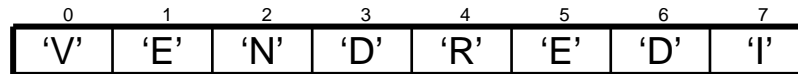
## □ Structure de données **linéaire**

### – Définition

- C'est une structure de données dans laquelle il existe un *ordre total strict* implicite entre ses éléments.
- Cela veut dire que, quels que soient deux éléments  $e_i$   $e_k$  rangés dans la structure, on peut dire quel est le « plus petit d'entre les deux » selon la relation la relation d'ordre sous-jacente.

### – Exemples

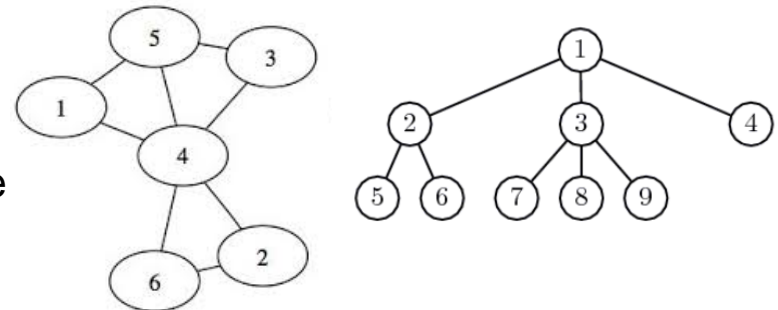
- Une chaîne de caractères, un tableau de caractères  
Les éléments sont ordonnés selon leur position de rangement dans la structure



- Une pile d'assiettes, une file d'attente devant le guichet de la poste  
Les éléments sont rangés par rapport à leur ordre d'arrivée dans la structure

### – Contre-exemples

- Les arbres et les graphes  
Il n'existe pas de relation d'ordre permettant de comparer les éléments entre eux



# 0.- Définitions préalables (5/5)

---

## □ Structure de données **statique** et **dynamique**

### – Définition Structure de données **statique**

C'est une structure de données dont la taille (occupation en mémoire) :


- est connue **avant** l'exécution du programme qui la manipule
- ne change pas au cours de l'exécution de ce programme

### – Définition Structure de données **dynamique**

C'est une structure de données dont la taille (occupation en mémoire) :

- n'est **pas** connue **avant** l'exécution du programme qui la manipule
- peut changer au cours de l'exécution de ce programme

# chapitre 4.- Utilisation de *Piles*



Ressource R1.01 : Initiation au développement - Partie 2

Institut Universitaire de Technologie de Bayonne – Pays Basque  
BUT Informatique – Semestre 1 - P. Dagorret



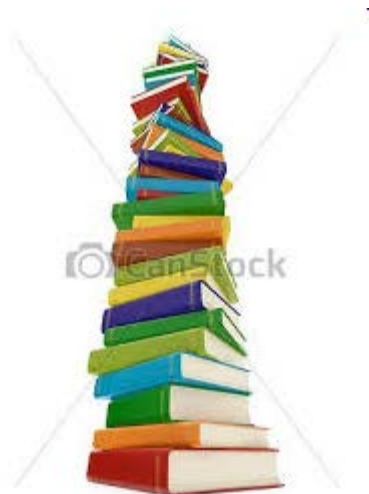
# 1.- Pile : Définition

---

## □ Définition **Pile**

- C'est une structure de données linéaire dans laquelle l'ordre implicite entre les éléments qu'elle contient est l'ordre ***chronologique d'entrée*** dans la structure
- Lorsque l'on consulte ou enlève un élément de la Pile, c'est toujours le ***dernier*** entré, c'est à dire le plus récent.
- Une Pile est une structure `LIFO` (*Last In First Out*), c'est à dire, *Dernier Entré, Premier Sorti*

## □ Illustration



© Can Stock Photo - csp12400824

La pile de livres :

- On ajoute un livre au-dessus de la pile
- La consultation ou le retrait ne peut se faire que sur le livre situé en haut de la pile, sinon on prend le risque de tout faire tomber....

# 1.- Pile : Définition

---

## Utilisation des piles en informatique

- appel de sous-programmes // empilage du contexte
- CTRL-Z (traitement de texte=)
- Backtracking (labyrinthe)
- À compléter

## 2.- Pile : Caractéristiques

---

### □ Domaine des valeurs

- Une pile est une structure homogène *éventuellement vide*
- Nous appellerons **UnePile** le type associé à cette structure
- Les variables piles manipulées dans nos programmes seront toutes de type `UnePile` :

```
// Déclaration d'une variable de type pile
UnePile maPileBD; // les BDs posées sur ma table de chevet
```

### □ Type des éléments

- Une pile est une structure homogène *générique* : les éléments pourront être d'un type quelconque
- Nous appellerons **UnElement** le type des éléments contenus dans cette structure

### □ Relation structurelle (ancienneté d'arrivée)

- L'ordre d'ancienneté d'arrivée dans la structure est une relation d'ordre totale et stricte :
  - Deux éléments sont toujours comparables par cette relation
  - L'ordre entre ces 2 éléments est strict

### 3.- Opérations possibles sur une Pile - Notation

---

- Une opération effectuée sur une pile peut affecter ou pas le contenu de celle-ci : la pile peut être, pour cette opération
  - une Donnée
  - un Résultat
  - Donnée et Résultat
- Convention de notation dans les pré et post conditions

Soit  $p$  une pile à laquelle sera appliquée une opération

  - Les propriétés de la pile  $p$  **avant** l'exécution de l'opération seront décrites à l'aide d'une ou plusieurs **pré-condition(s)**
  - Les propriétés de la pile **après** l'exécution de l'opération seront décrites à l'aide d'une ou plusieurs **post-condition(s)**. Dans la post-condition, la pile sera désignée par  $p'$  (le nom de l'élément suivi d'une apostrophe simple)
  - Exemple : Ajouter un élément dans la pile  $p$  :
    - pré-condition :  $p$  n'est pas pleine
    - post-condition :  $p'$  n'est pas vide

### 3.- Opérations possibles sur une Pile - Initialiser une Pile (1/2)

---

- Nom de la primitive : **initialiser**
- But : Met à disposition une pile prête à l'emploi VIDE
- Données : une pile  $p$ 
  - pré-condition :  $\phi$
- Résultat :  $p$ 
  - post-condition : `estVide(p') = vrai`
- Entête de la primitive :

```
void initialiser (UnePile& p);
```

- Exemple d'appel :

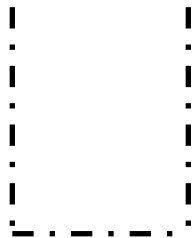
```
initialiser(maPile);
```

### 3.- Opérations possibles sur une Pile - Initialiser une Pile (2/2)

---

**initialiser**(maPile)

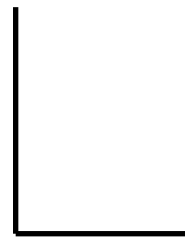
État de la pile **avant** l'opération



maPile

Pile déclarée

État de la pile **après** l'opération

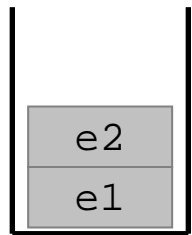


maPile

Pile vide, prête à l'emploi

**initialiser**(maPile)

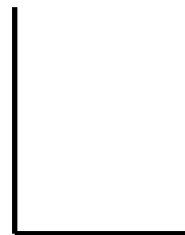
État de la pile **avant** l'opération



maPile

Pile déjà utilisée

État de la pile **après** l'opération



maPile

Pile vide, prête à l'emploi

### 3.- Opérations possibles sur une Pile - Déterminer si une Pile est vide (1/2)

---

- Nom de la primitive : **estVide**
- But : Retourne **vrai** si la pile ne contient aucun élément, **faux** sinon
- Données : une pile  $p$ 
  - pré-condition :  $\phi$
- Valeur résultante : un booléen
  - post-condition :  $\phi$
- Entête de la primitive :

```
bool estVide (const UnePile& p);
```

- Exemple d'appel :

```
if (estVide(maPile)) ...
```

### 3.- Opérations possibles sur une Pile -

## Déterminer si une Pile est vide (2/2)

---

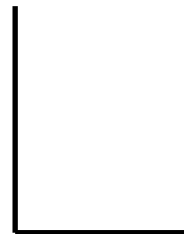
**estVide**(maPile)

État de la pile **avant** l'opération



maPile

État de la pile **après** l'opération

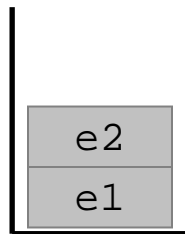


maPile

**retourne vrai**

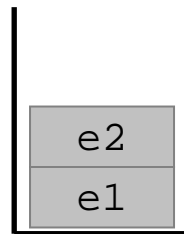
**estVide**(maPile)

État de la pile **avant** l'opération



maPile

État de la pile **après** l'opération



maPile

**retourne faux**



### 3.- Opérations possibles sur une Pile -

## Déterminer si une Pile est pleine (1/2)

---

- Nom de la primitive : **estPleine**
- But : Retourne **vrai** si la pile ne peut plus stocker d'élément, **faux** sinon
- Données : une pile  $p$ 
  - pré-condition :  $\phi$
- Valeur résultante : un booléen
  - post-condition :  $\phi$

- Entête de la primitive :

```
bool estPleine (const UnePile& p);
```

- Exemple d'appel :

```
if (estPleine(maPile)) ...
```

- Attention : Primitive en général **non** disponible dans une implantation dynamique

### 3.- Opérations possibles sur une Pile - Déterminer si une Pile est pleine (2/2)

---

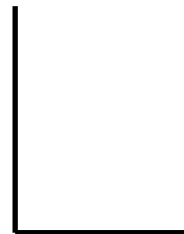
**estPleine**(maPile)

État de la pile **avant** l'opération



maPile

État de la pile **après** l'opération

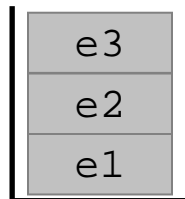


maPile

**retourne faux**

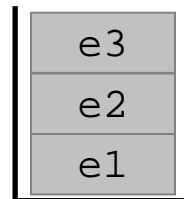
**estPleine**(maPile)

État de la pile **avant** l'opération



maPile

État de la pile **après** l'opération



maPile

**retourne vrai**

### 3.- Opérations possibles sur une Pile -

## Déterminer le nombre d'éléments d'une Pile (1/2)

---

- Nom de la primitive : **taille**
- But : Retourne le nombre d'éléments stockés dans la pile
- Données : une pile  $p$ 
  - pré-condition :  $\phi$
- Valeur résultante : un entier naturel
  - post-condition :  $\phi$
- Entête de la primitive :

```
unsigned int taille (const UnePile& p);
```

- Exemple d'appel :

```
if (taille(maPile) > 15) ...
```

### 3.- Opérations possibles sur une Pile -

## Déterminer le nombre d'éléments d'une Pile (2/2)

---

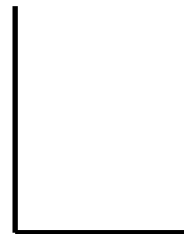
**taille**(maPile)

État de la pile **avant** l'opération



maPile

État de la pile **après** l'opération

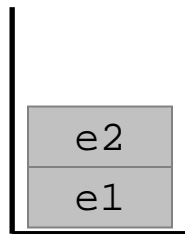


maPile

retourne 0

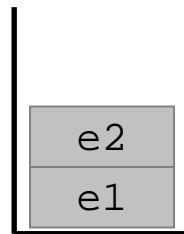
**taille**(maPile)

État de la pile **avant** l'opération



maPile

État de la pile **après** l'opération



maPile

retourne 2

### 3.- Opérations possibles sur une Pile -

## Connaître la valeur du sommet de la Pile (1/2)

---

- Nom de la primitive : **sommet**
- But : Retourne la valeur de l'élément situé au sommet de la pile
- Données : une pile *p*
  - pré-condition : `estVide(p) = faux`  
Génère l'exception "pileVide" sinon
- Valeur résultante : une valeur de type **UnElement**
  - post-condition :  $\phi$
- Entête de la primitive :

```
UnElement sommet (const UnePile& p);
```

- Exemple d'appel :

```
UnElement lePlusHaut;  
lePlusHaut = sommet(maPile);
```

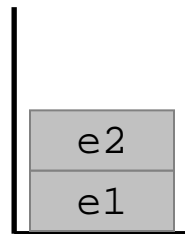
### 3.- Opérations possibles sur une Pile -

## Connaître la valeur du sommet de la Pile (2/2)

---

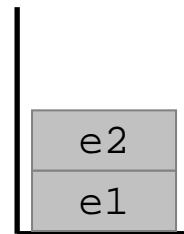
**sommet**(maPile)

État de la pile **avant** l'opération



maPile

État de la pile **après** l'opération



maPile

retourne la valeur e2

**sommet**(maPile)

État de la pile **avant** l'opération



maPile

État de la pile **après** l'opération



maPile

exception "pileVide"  
levée

### 3.- Opérations possibles sur une Pile -

## Ajouter un élément au sommet de la Pile (1/2)

---

- Nom de la primitive : **empiler**
- But : Ajoute un élément au sommet de la pile
- Données : une pile  $p$ , un élément  $e$  à ajouter sur  $p$ 
  - pré-condition : `estPleine(p) = faux`  
Génère l'exception "pilePleine" sinon
- Résultat : la pile  $p$  modifiée
  - post-condition : `estVide(p') = faux` **et** `sommet(p') = e`
- Entête de la primitive :

```
void empiler (UnePile& p, UnElement e);
```

- Exemple d'appel :

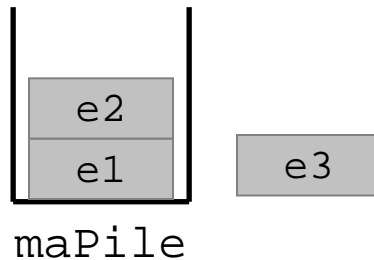
```
empiler(maPile, 5);
```

### 3.- Opérations possibles sur une Pile -

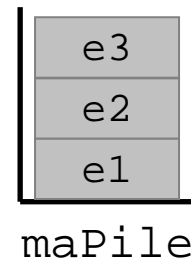
## Ajouter un élément au sommet de la Pile (2/2)

`empiler(maPile, e3)`

État de la pile **avant** l'opération

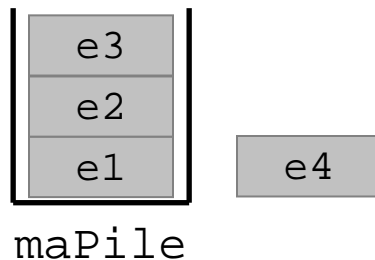


État de la pile **après** l'opération

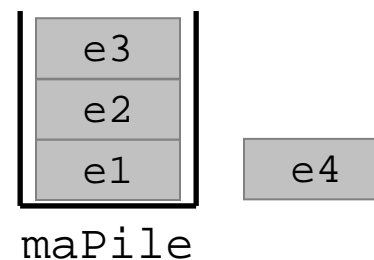


`empiler(maPile, e4)`

État de la pile **avant** l'opération



État de la pile **après** l'opération



exception "pilePeine"  
levée



### 3.- Opérations possibles sur une Pile -

## Supprimer l'élément situé au sommet de la Pile (1/2)

---

- Nom de la primitive : **dépiler**
- But : Supprime l'élément situé au sommet de la pile
- Données : une pile  $p$ 
  - pré-condition : `estVide(p) = faux`  
Génère l'exception "pileVide" sinon
- Résultat : la pile  $p$  modifiée
  - post-condition : `estPleine(p') = faux`
- Entête de la primitive :

```
void depiler (UnePile& p);
```

- Exemple d'appel :

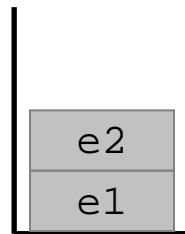
```
depiler(maPile);
```

### 3.- Opérations possibles sur une Pile -

## Supprimer l'élément situé au sommet de la Pile (2/2)

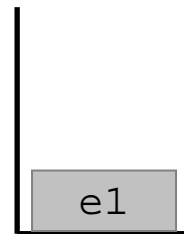
**depiler**(maPile)

État de la pile **avant** l'opération



maPile

État de la pile **après** l'opération



maPile

**depiler**(maPile)

État de la pile **avant** l'opération



maPile

État de la pile **après** l'opération



maPile

**exception "pileVide"  
levée**

### 3.- Opérations possibles sur une Pile - *variante de* Supprimer l'élément situé au sommet de la Pile (1/2)

---

- Nom de la primitive : **depiler**
- But : Supprime l'élément situé au sommet de la pile et le range dans une variable
- Données : une pile *p*
  - pré-condition : `estVide(p) = faux`  
Génère l'exception "pileVide" sinon
- Résultante : la pile *p* modifiée, l'élément *e* qui était au sommet de la pile
  - post-condition : `estPleine(p') = faux` **et** `e' = sommet(p)`

- Entête de la primitive :

```
void depiler (UnePile& p, UnElement& e);
```

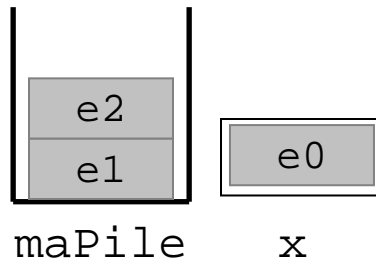
- Exemple d'appel :

```
UnElement x;  
depiler(maPile, x);
```

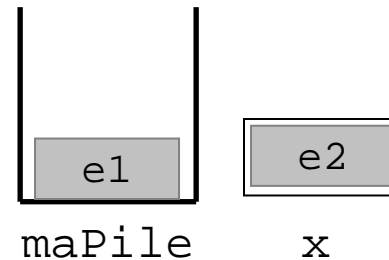
### 3.- Opérations possibles sur une Pile - *variante de* Supprimer l'élément situé au sommet de la Pile (2/2)

`depiler`(maPile, x)

État de la pile **avant** l'opération

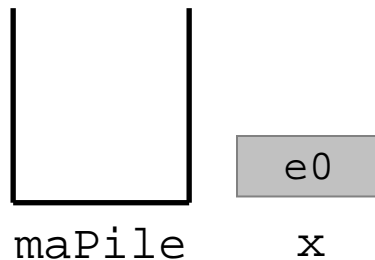


État de la pile **après** l'opération

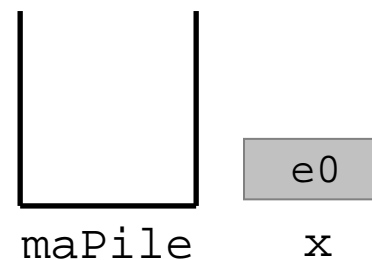


`depiler`(maPile, x)

État de la pile **avant** l'opération



État de la pile **après** l'opération



**exception "pileVide"  
levée**

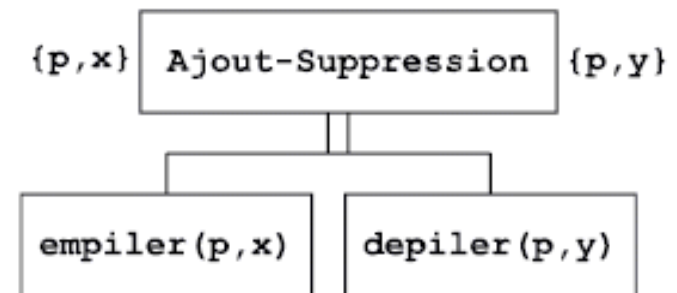
## 4.- Ajouts et Suppressions d'éléments sur une Pile : Synthèse

### □ A retenir

- La seule façon d'ajouter un élément dans une pile est de l'ajouter au sommet de la pile
- La seule façon de supprimer un élément dans une pile est de supprimer celui situé sur le sommet de la pile

### □ Exercice

Étant donné l'algorithme suivant :



- Écrire les propriétés (post-conditions) de  $y$  et  $p$  suite à l'exécution de l'algorithme
- A quelle instruction est équivalent cet algorithme ?

## 5.- Erreurs liées à l'utilisation d'une structure de données avancée (1/2)

### □ Postulat :

La mauvaise utilisation d'une structure de données fait partie intégrante de l'usage de cette structure de données

### □ Ainsi, le **concepteur** d'une structure de données avancée doit

- Définir et décrire les primitives de manipulation de la structure de données

- Identifier les problèmes/erreurs pouvant surgir suite à un mauvais usage de chaque primitive

- Préciser en conséquence les conditions d'utilisation de chaque primitive, **sous forme de pré-conditions**

- Alerter, sous la forme d'une levée d'exception, lors de chaque mauvaise/erreur d'utilisation de la structure de données.

▲ Nom de la primitive : **sommet**

▲ But : Retourne la valeur de l'élément situé au sommet de la pile

▲ Données : une pile p

▼ pré-condition : `estVide(p) = faux`

▼ Génère l'exception "pilevide" sinon

▲ Valeur résultante : une valeur de type **UnElement**

— post-condition :  $\phi$

▲ Entête de la primitive :

```
UnElement sommet (const UnePile& p);
```

▲ Exemple d'appel :

```
UnElement lePlusHaut;  
lePlusHaut = sommet (maPile);
```

Pour le concepteur d'une structure de données, **toute primitive soumise à une pré-condition suppose donc le déclenchement d'une erreur** lorsque la pré-condition signalée n'est pas respectée

## 5.- Erreurs liées à l'utilisation d'une structure de données avancée (2/2)

---

- De son côté, le **programmeur utilisant** une structure de données avancée doit
  - Identifier et comprendre les conditions d'utilisation de chaque primitive,
  - **Intégrer un test de vérification de la pré-condition** dans l'algorithme qui utilise la primitive.

Les erreurs déclenchées doivent être perçues comme une **aide** fournie au programmeur pour améliorer son code en le consolidant face aux imprévus.

Pour ce faire, les erreurs doivent être décrites par le concepteur selon un canevas précis :

- Schéma descriptif d'une erreur
  - Un nom, permettant de l'identifier de manière claire
  - Une description du contexte d'utilisation de la structure de données qui fait déclencher l'erreur
  - La liste des primitives susceptibles de déclencher l'erreur

## 6.- Pile : Erreurs liées à son utilisation

---

- Nom de l'erreur : **"pileVide"**
- Condition de déclenchement  
Lorsque l'on tente d'accéder au sommet de la pile alors que celle-ci est vide
- Opérations susceptibles de la déclencher
  - sommet
  - depiler
- Nom de l'erreur : **"pilePleine"**
- Condition de déclenchement  
Lorsque l'on tente d'ajouter un élément au sommet de la pile alors que celle-ci est pleine
- Opérations susceptibles de la déclencher
  - empiler



## 7.- Code C++ d'un Gestionnaire de Piles :

### Définition du type des éléments contenus dans la Pile

---

```
1  #ifndef PILE_H
2  #define PILE_H
3
4  #include <stack>
5  using namespace std;
6
7  // DEFINITION DU TYPE DES ELEMENTS CONTENUS DANS LA PILE
8
9  /*   Pour faire une pile avec des types de base :
10     typedef int UnElement; */
11
12  /*   Pour faire une pile avec des elements de type struct :
13     typedef struct
14     {
15         int coordX; // abscisse du point
16         int coordY; // ordonnee du point
17     } UnElement; */
18
19  /*   Pour faire une pile avec des elements dont le type est defini dans un autre
20     fichier :
21     #include "leFichierOuEstDefiniLeType.h"
22     typedef leTypeDefiniDansLeFichier UnElement; */
23
24  typedef int UnElement;
25
26
27  // DEFINITION DE LA PILE
28  typedef stack<UnElement> UnePile;
```

## 7.- Code C++ d'un Gestionnaire de Piles : Primitives disponibles

---

```
30 // PRIMITIVES
31
32 void initialiser (UnePile& p);
33 // Initialise ou ré-initialise une pile vide prête à l'emploi
34
35 unsigned int taille (const UnePile& p);
36 // Retourne le nombre d'elements contenus dans la pile p
37
38 bool estVide (const UnePile& p);
39 // Retourne vrai si la pile p est vide, faux sinon
40
41 UnElement sommet (const UnePile& p);
42 /* Retourne l'element situe au sommet de la pile p
43    Genere l'exception "pileVide" si la pile p est vide */
44
45 void empiler (UnePile& p, UnElement e);
46 // Ajoute l'element e au sommet de ma pile p
47
48 void depiler (UnePile& p);
49 /* Retire l'element situe au sommet de la pile p
50    Genere l'exception "pileVide" si la pile p est vide */
51
52 void depiler (UnePile& p, UnElement& e);
53 /* Retire l'element situe au sommet de la pile p et le stocke dans e
54    Genere l'exception "pileVide" si la pile p est vide */
55
56 #endif
```

## 8.- Utilisation d'une Pile :

### Exemple de code C++ utilisant une pile d'entiers

```
1  #include <iostream>
2  using namespace std;
3  #include "pile.h"
4
5  int main()
6  {
7      //Déclaration et initialisation de la pile
8      UnePile maPile;
9      initialiser(maPile);
10
11     // Ajout d'éléments dans la pile
12     empiler(maPile, 0); empiler(maPile, 1);
13     empiler(maPile, 2); empiler(maPile, 3);
14     empiler(maPile, 4); empiler(maPile, 5);
15
16     // Affichage du contenu de la pile
17     cout << sommet(maPile) << "... " ; depiler(maPile);
18     cout << sommet(maPile) << "... " ; depiler(maPile);
19     cout << sommet(maPile) << "... " ; depiler(maPile);
20     cout << sommet(maPile) << "... " ; depiler(maPile);
21     cout << sommet(maPile) << "... " ; depiler(maPile);
22     cout << sommet(maPile) << "... " ; depiler(maPile);
23
24     return 0;
25 }
```

□ Affichage produit :

□ En ligne 23 (mode Debug), que valent les expressions

`estVide(maPile)?`

`estPleine(maPile)?`

`taille(maPile)?`

## 8.- Utilisation d'une Pile :

### Exemple de code C++ utilisant une pile d'entiers

```
1  #include <iostream>
2  using namespace std;
3  #include "pile.h"
4
5  int main()
6  {
7      //Déclaration et initialisation de la pile
8      UnePile maPile;
9      initialiser(maPile);
10
11     // Ajout d'éléments dans la pile
12     empiler(maPile, 0); empiler(maPile, 1);
13     empiler(maPile, 2); empiler(maPile, 3);
14     empiler(maPile, 4); empiler(maPile, 5);
15
16     // Affichage du contenu de la pile
17     cout << sommet(maPile) << "... " ; depiler(maPile);
18     cout << sommet(maPile) << "... " ; depiler(maPile);
19     cout << sommet(maPile) << "... " ; depiler(maPile);
20     cout << sommet(maPile) << "... " ; depiler(maPile);
21     cout << sommet(maPile) << "... " ; depiler(maPile);
22     cout << sommet(maPile) << "... " ; depiler(maPile);
23
24     return 0;
25 }
```

□ Affichage produit : 5... 4... 3... 2... 1... 0...

□ En ligne 23 (mode Debug), que valent les expressions

`estVide(maPile)?` **TRUE**

`estPleine(maPile)?` **FALSE**

`taille(maPile)?` **0**

## 8.- Utilisation d'une Pile :

### Exemple de code C++ utilisant une pile d'entiers

- But du code : Afficher **tout** le contenu de la pile

```
16 // Affichage du contenu de la pile
17 cout << sommet(maPile) << "... " ; depiler(maPile);
18 cout << sommet(maPile) << "... " ; depiler(maPile);
19 cout << sommet(maPile) << "... " ; depiler(maPile);
20 cout << sommet(maPile) << "... " ; depiler(maPile);
21 cout << sommet(maPile) << "... " ; depiler(maPile);
22 cout << sommet(maPile) << "... " ; depiler(maPile);
```

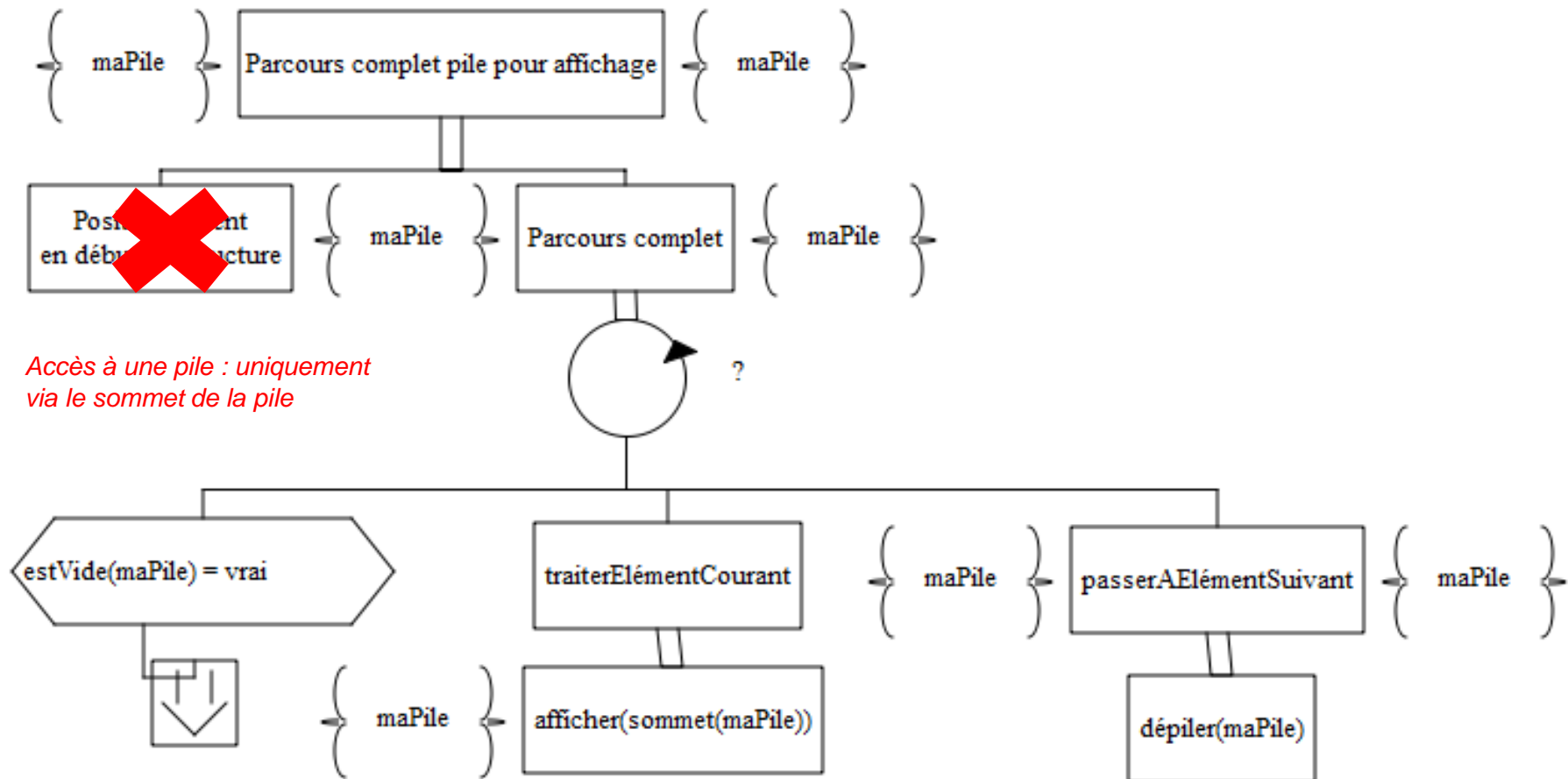
#### Modèle d'algorithme

= Parcours complet de la pile avec traitement systématique  
Traitement à répéter : afficher élément courant

- Une pile est une structure de données à accès séquentiel !  
→ voir Chapitre 2  
§3.- Parcours séquentiel complet avec traitement systématique

## 8.- Utilisation d'une Pile :

### Modèle d'algorithme associé



## 8.- Utilisation d'une Pile :

### Exemple de code C++ utilisant une pile d'entiers

```
1  #include<iostream>
2  using namespace std;
3  #include "pile.h"
4
5  int main(void)
6  {
7      // Déclaration et initialisation de la pile
8      UnePile maPile;
9      initialiser(maPile);
10
11     // Ajout d'éléments dans la pile
12     empiler (maPile, 0); empiler (maPile, 1);
13     empiler (maPile, 2); empiler (maPile, 3);
14     empiler (maPile, 4); empiler (maPile, 5);
15
16     // Affichage du contenu de la pile
17     cout << "Compte a rebours ..." << endl;
18     while ( !estVide(maPile) )
19     {
20         cout << sommet(maPile) << "... ";
21         depiler(maPile);
22     }
23
24     return 0;
25 }
```

□ Affichage produit : 5... 4... 3... 2... 1... 0...


□ En ligne 23 (mode Debug), que valent les expressions

`estVide(maPile)?` **TRUE**

`estPleine(maPile)?` **FALSE**

`taille(maPile)?` **0**

# chapitre 4.- Utilisation de *Files*



Ressource R1.01 : Initiation au développement - Partie 2

Institut Universitaire de Technologie de Bayonne – Pays Basque  
BUT Informatique – Semestre 1 - P. Dagorret



# 1.- File : Définition

---

## □ Définition **File**

- C'est une structure de données linéaire dans laquelle l'ordre implicite entre les éléments qu'elle contient est l'ordre ***chronologique d'entrée*** dans la structure
- Lorsque l'on consulte ou enlève un élément de la File, c'est toujours le ***premier*** entré, c'est à dire le plus ancien.
- Une File est une structure **FIFO** (*First In First Out*), c'est à dire, *Premier Entré, Premier Sorti*

## □ Illustrations



La file d'attente devant un guichet, un distributeur d'argent, ... :

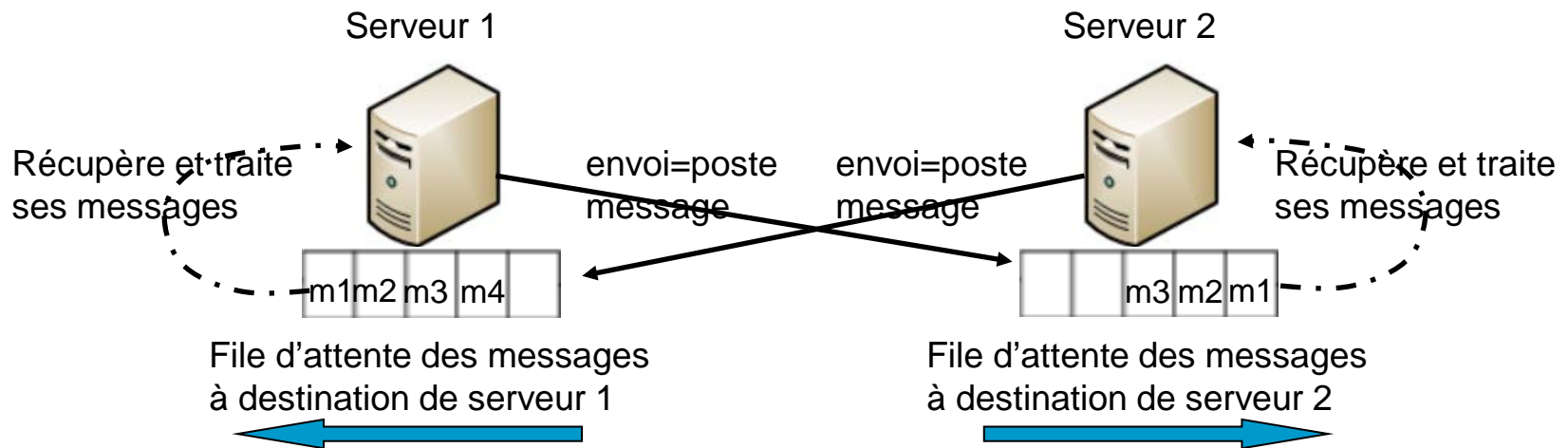
- La personne qui est servie est située en **tête** de file, c'est la première personne arrivée
- La **dernière** personne arrivée est en **fin** (ou **queue**) de file. Elle attend d'arriver en tête pour pouvoir être servie....

# 1.- File : Définition

## □ Illustrations dans le domaine informatique

### – File d'attente de messages

Elles permettent le fonctionnement des liaisons asynchrones entre deux serveurs, c'est-à-dire de canaux de communications tels que l'expéditeur et le récepteur du message ne soient pas contraints de s'attendre l'un l'autre, mais poursuivent chacun l'exécution de leurs tâches.



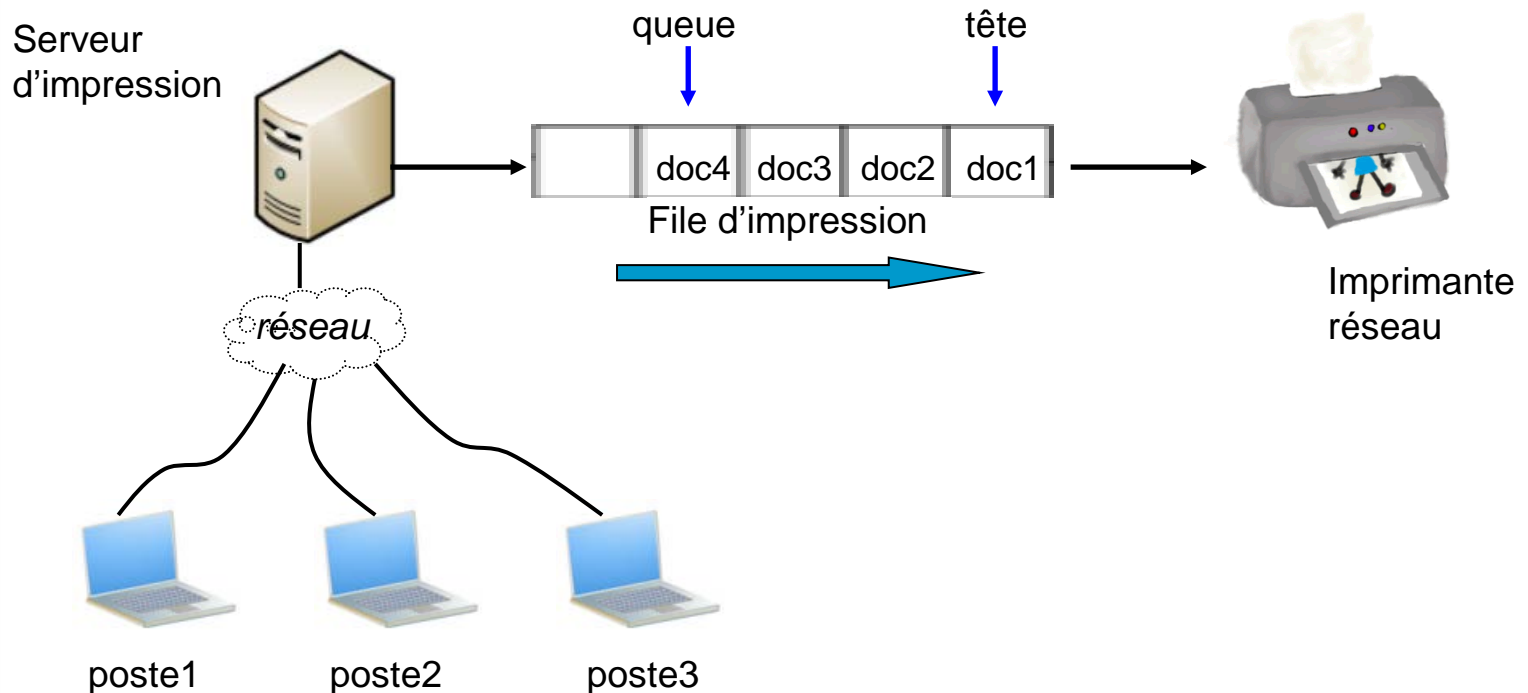
La file d'attente stocke les messages émis par l'expéditeur, jusqu'à ce que le destinataire les recherche. L'expéditeur n'a pas à attendre que le récepteur commence à traiter son message, il poste son information et peut passer à autre chose.

# 1.- File : Définition

## □ Illustrations dans le domaine informatique

### – File d'attente d'impression

Différents postes de travail envoient par le réseau leur fichiers à imprimer. Un serveur d'impression réceptionne les fichiers et les stocke dans une file d'attente, le **spooler**. Les travaux d'impression sont envoyés à l'imprimante selon l'ordre d'arrivée dans la file d'attente.



## 2.- File : Caractéristiques

---

### □ Domaine des valeurs

- Une file est une structure homogène *éventuellement vide*
- Nous appellerons **UneFile** le type associé à cette structure
- Les variables files manipulées dans nos programmes seront toutes de type `UneFile` :

```
// Déclaration d'une variable de type file  
UneFile maFileImpression;
```

### □ Type des éléments

- Une file est une structure homogène *générique* : les éléments pourront être d'un type quelconque
- Nous appellerons **UnElement** le type des éléments contenus dans cette structure

### □ Relation structurelle (ancienneté d'arrivée)

- L'ordre d'ancienneté d'arrivée dans la structure est une relation d'ordre totale et stricte :
  - Deux éléments sont toujours comparables par cette relation
  - L'ordre entre ces 2 éléments est strict

### 3.- Opérations possibles sur une File - Notation

---

- Une opération effectuée sur une file peut affecter ou pas le contenu de celle-ci : la file peut être, pour cette opération
  - une Donnée
  - un Résultat
  - Donnée et Résultat
- Convention de notation dans les pré et post conditions

Soit  $f$  une file à laquelle sera appliquée une opération

  - Les propriétés de la file  $f$  **avant** l'exécution de l'opération seront décrites à l'aide d'une ou plusieurs **pré-condition(s)**
  - Les propriétés de la file **après** l'exécution de l'opération seront décrites à l'aide d'une ou plusieurs **post-condition(s)**. Dans la post-condition, la file sera désignée par  $f'$  (le nom de l'élément suivi d'une apostrophe simple)
  - Exemple : Ajouter un élément dans la file  $f$  :
    - pré-condition :  $f$  n'est pas pleine
    - post-condition :  $f'$  n'est pas vide

### 3.- Opérations possibles sur une File - Initialiser une File (1/2)

---

- Nom de la primitive : **initialiser**
- But : Met à disposition une file prête à l'emploi VIDE
- Données : une file  $f$ 
  - pré-condition :  $\phi$
- Résultat :  $f$ 
  - post-condition : `estVide(f') = vrai`
- Entête de la primitive :

```
void initialiser (UneFile& f);
```

- Exemple d'appel :

```
initialiser(maFile);
```

### 3.- Opérations possibles sur une File - Initialiser une File (2/2)

**initialiser**(maFile)

État de la file **avant** l'opération

queue tête

-----

-----

maFile File déclarée

État de la file **après** l'opération

queue tête

=====

=====

maFile File vide, prête à l'emploi

**initialiser**(maFile)

État de la file **avant** l'opération

queue tête

=====

e2 e1

=====

maFile File déjà utilisée

État de la file **après** l'opération

queue tête

=====

=====

maFile File vide, prête à l'emploi

### 3.- Opérations possibles sur une File - Déterminer si une File est vide (1/2)

---

- Nom de la primitive : **estVide**
- But : Retourne **vrai** si la file ne contient aucun élément, **faux** sinon
- Données : une file  $f$ 
  - pré-condition :  $\phi$
- Valeur résultante : un booléen
  - post-condition :  $\phi$
- Entête de la primitive :

```
bool estVide (const UneFile& f);
```

- Exemple d'appel :

```
if (estVide(maFile)) ...
```



### 3.- Opérations possibles sur une File - Déterminer si une File est vide (2/2)

**estVide**(maFile)

État de la file **avant** l'opération

queue tête

\_\_\_\_\_

maFile

État de la file **après** l'opération

queue tête

\_\_\_\_\_

maFile

retourne vrai

**estVide**(maFile)

État de la file **avant** l'opération

queue tête

\_\_\_\_\_

e2 e1

maFile

État de la file **après** l'opération

queue tête

\_\_\_\_\_

e2 e1

maFile

retourne faux

### 3.- Opérations possibles sur une File -

## Déterminer si une File est pleine (1/2)

---

- Nom de la primitive : **estPleine**
- But : Retourne **vrai** si la file ne peut plus stocker d'élément, **faux** sinon
- Données : une file  $f$ 
  - pré-condition :  $\phi$
- Valeur résultante : un booléen
  - post-condition :  $\phi$

- Entête de la primitive :

```
bool estPleine (const UneFile& f);
```

- Exemple d'appel :

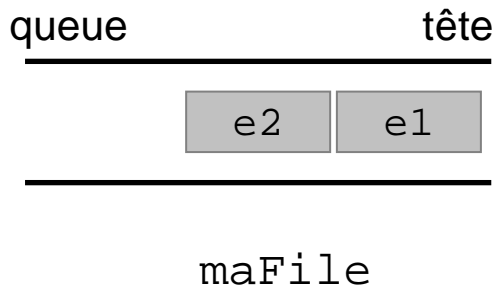
```
if (estPleine(maFile)) ...
```

- Attention : Primitive en général **non** disponible dans une implantation dynamique

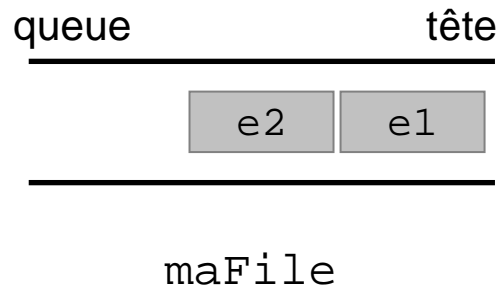
### 3.- Opérations possibles sur une File - Déterminer si une File est pleine (2/2)

**estPleine**(maFile)

État de la file **avant** l'opération



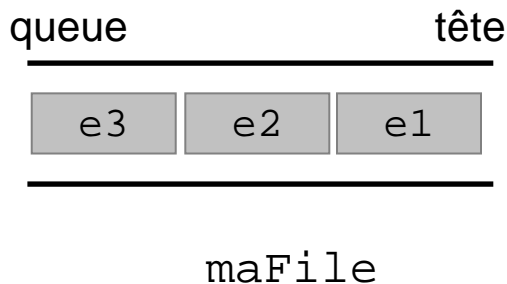
État de la file **après** l'opération



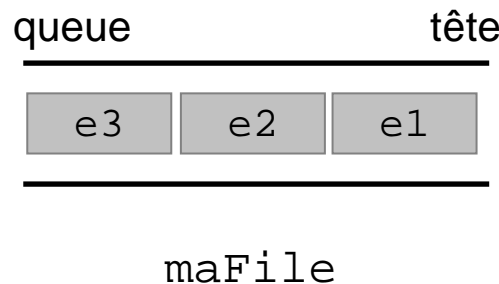
retourne faux

**estPleine**(maFile)

État de la file **avant** l'opération



État de la file **après** l'opération



retourne vrai

### 3.- Opérations possibles sur une File -

## Déterminer le nombre d'éléments d'une File (1/2)

---

- Nom de la primitive : **taille**
- But : Retourne le nombre d'éléments stockés dans la file
- Données : une file  $f$ 
  - pré-condition :  $\phi$
- Valeur résultante : un entier naturel
  - post-condition :  $\phi$
- Entête de la primitive :

```
unsigned int taille (const UneFile& f);
```

- Exemple d'appel :

```
if (taille(maFile) > 15) ...
```

### 3.- Opérations possibles sur une File -

## Déterminer le nombre d'éléments d'une File (2/2)

`taille`(maFile)

État de la file **avant** l'opération

queue tête

\_\_\_\_\_

maFile

État de la file **après** l'opération

queue tête

\_\_\_\_\_

maFile

retourne 0

`taille`(maFile)

État de la file **avant** l'opération

queue tête

\_\_\_\_\_

e2 e1

maFile

État de la file **après** l'opération

queue tête

\_\_\_\_\_

e2 e1

maFile

retourne 2

### 3.- Opérations possibles sur une File -

## Connaître la valeur de la tête de la File (1/2)

---

- Nom de la primitive : **premier**
- But : Retourne la valeur de l'élément situé en tête de file
- Données : une file  $f$ 
  - pré-condition : `estVide(f) = faux`  
Génère l'exception "fileVide" sinon
- Valeur résultante : une valeur de type **UnElement**
  - post-condition :  $\phi$
- Entête de la primitive :

```
UnElement premier (const UnFile& f);
```

- Exemple d'appel :

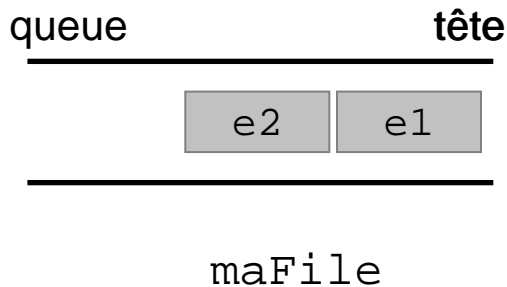
```
UnElement lePremier;  
lePremier = premier(maFile);
```

### 3.- Opérations possibles sur une File -

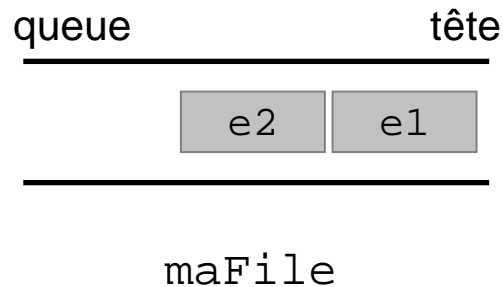
## Connaître la valeur de la tête de la File (2/2)

`premier`(maFile)

État de la file **avant** l'opération



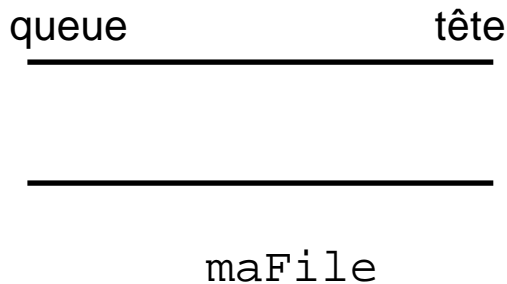
État de la file **après** l'opération



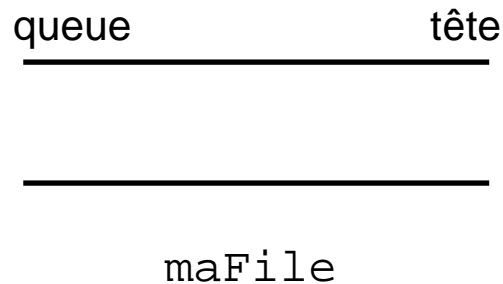
retourne  
la valeur e1

`premier`(maFile)

État de la file **avant** l'opération



État de la file **après** l'opération



exception  
"fileVide" levée

### 3.- Opérations possibles sur une File -

## Ajouter un élément en fin=queue de File (1/2)

---

- Nom de la primitive : **enfiler**
- But : Ajoute un élément en fin (= en queue) de la file
- Données : une file  $f$ , un élément  $e$  à ajouter en fin de  $f$ 
  - pré-condition : `estPleine(f) = faux`  
Génère l'exception "filePleine" sinon
- Résultat : la file  $f$  modifiée
  - post-condition : `estVide(f') = faux`
- Entête de la primitive :

```
void enfiler (UneFile& f, UnElement e);
```

- Exemple d'appel :

```
enfiler(maFile, 5);
```

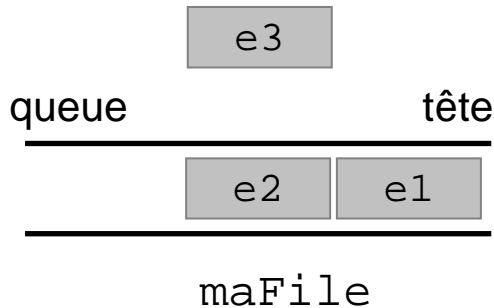


### 3.- Opérations possibles sur une File -

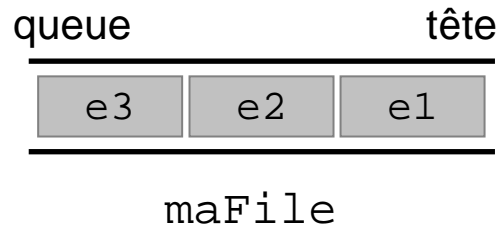
## Ajouter un élément en fin=queue de File (2/2)

**enfiler**(maFile, e3)

État de la file **avant** l'opération

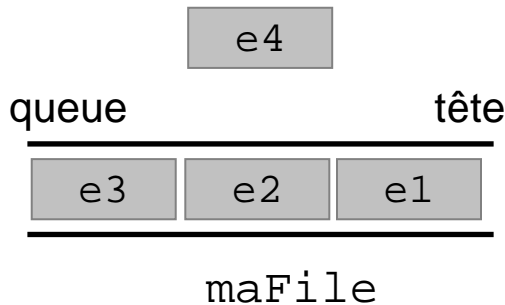


État de la file **après** l'opération

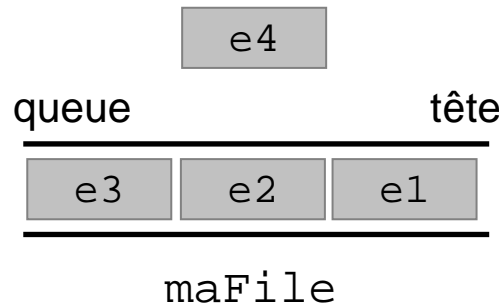


**enfiler**(maFile, e4)

État de la file **avant** l'opération



État de la file **après** l'opération



**exception**  
**"filePeine" levée**

### 3.- Opérations possibles sur une File -

## Supprimer l'élément situé en tête de File (1/2)

---

- Nom de la primitive : **defiler**
- But : Supprime l'élément situé en tête de file
- Données : une file  $f$ 
  - pré-condition : `estVide(f) = faux`  
Génère l'exception "fileVide" sinon
- Résultat : la file  $f$  modifiée
  - post-condition : `estPleine(f') = faux`
- Entête de la primitive :

```
void defiler (UneFile& f);
```

- Exemple d'appel :

```
defiler(maFile);
```

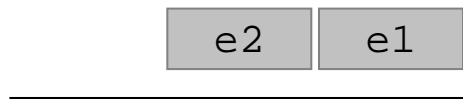
### 3.- Opérations possibles sur une File -

## Supprimer l'élément situé en tête de File (2/2)

**defiler**(maFile)

État de la file **avant** l'opération

queue tête



maFile

État de la file **après** l'opération

queue tête



maFile

**defiler**(maFile)

État de la file **avant** l'opération

queue tête



maFile

État de la file **après** l'opération

queue tête



maFile

**exception  
"fileVide" levée**

### 3.- Opérations possibles sur une File - *variante de* Supprimer l'élément situé en tête de File (1/2)

---

- Nom de la primitive : **defiler**
- But : Supprime l'élément situé en tête de file et le range dans une variable
- Données : une file  $f$ 
  - pré-condition : `estVide(f) = faux`  
Génère l'exception "fileVide" sinon
- Résultante : la file  $f$  modifiée, l'élément  $e$  qui était en tête de file
  - post-condition : `estPleine(f') = faux` **et** `e' = premier(f)`
- Entête de la primitive :

```
void defiler (UneFile& f, UnElement& e);
```

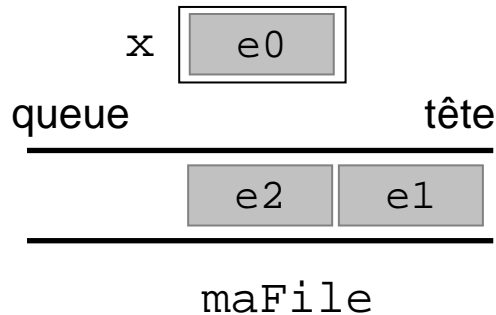
- Exemple d'appel :

```
UnElement x;  
defiler(maFile, x);
```

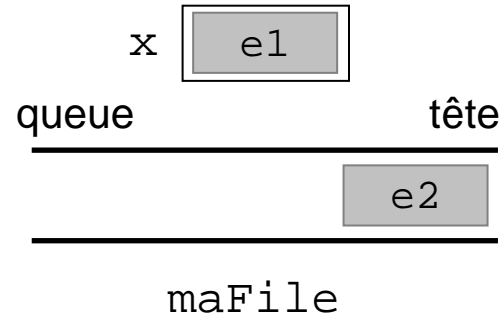
### 3.- Opérations possibles sur une File - *variante de* Supprimer l'élément situé en tête de File (2/2)

`defiler`(maFile, x)

État de la file **avant** l'opération

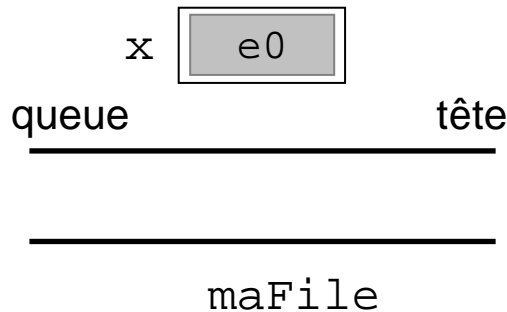


État de la file **après** l'opération

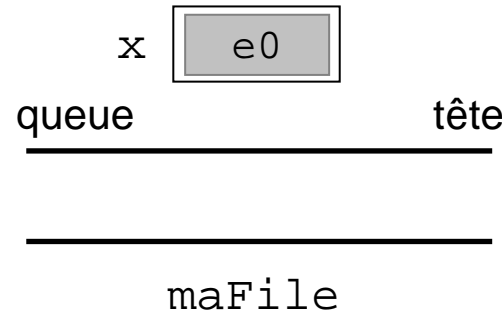


`defiler`(maFile, x)

État de la file **avant** l'opération



État de la file **après** l'opération



**exception  
"fileVide" levée**

## 4.- Ajout & Suppression d'éléments sur une File : Synthèse

---

### □ A retenir

- La seule façon d'ajouter un élément dans une file est de l'ajouter en queue=fin de file
- La seule façon de supprimer un élément dans une file est de supprimer celui situé en tête de file

## 5.- Erreurs liées à l'utilisation d'une structure de données avancée (1/2)

### □ Postulat :

La mauvaise utilisation d'une structure de données fait partie intégrante de l'usage de cette structure de données

### □ Ainsi, le **concepteur** d'une structure de données avancée doit

- Définir et décrire les primitives de manipulation de la structure de données

- Identifier les problèmes/erreurs pouvant surgir suite à un mauvais usage de chaque primitive

- Préciser en conséquence les conditions d'utilisation de chaque primitive, **sous forme de pré-conditions**

- Alerter, sous la forme d'une levée d'exception, lors de chaque mauvaise/erreur d'utilisation de la structure de données.

▲ Nom de la primitive : **premier**  
▲ But : Retourne la valeur de l'élément situé en tête de file  
▲ Données : une file *f*  
▼ pré-condition : `estVide(f) = faux`  
▼ Génère l'exception "fileVide" sinon

▲ Valeur résultante : une valeur de type **UnElement**  
– post-condition :  $\phi$

▲ Entête de la primitive :

```
UnElement premier (const UneFile& f);
```

▲ Exemple d'appel :

```
UnElement lePremier;  
lePremier = premier(maFile);
```

Pour le concepteur d'une structure de données, **toute primitive soumise à une pré-condition suppose donc le déclenchement d'une erreur** lorsque la pré-condition signalée n'est pas respectée

## 5.- Erreurs liées à l'utilisation d'une structure de données avancée (2/2)

---

- De son côté, le **programmeur utilisant** une structure de données avancée doit
  - Identifier et comprendre les conditions d'utilisation de chaque primitive,
  - **Intégrer un test de vérification de la pré-condition** dans l'algorithme qui utilise la primitive.

Les erreurs déclenchées doivent être perçues comme une **aide** fournie au programmeur pour améliorer son code en le consolidant face aux imprévus.

Pour ce faire, les erreurs doivent être décrites par le concepteur selon un canevas précis :

- Schéma descriptif d'une erreur
  - Un nom, permettant de l'identifier de manière claire
  - Une description du contexte d'utilisation de la structure de données qui fait déclencher l'erreur
  - La liste des primitives susceptibles de déclencher l'erreur



## 6.- File : Erreurs liées à son utilisation

---

- ❑ Nom de l'erreur : **"fileVide"**
- ❑ Condition de déclenchement  
Lorsque l'on tente d'accéder à l'élément situé tête de file alors que celle-ci est vide
- ❑ Opérations susceptibles de la déclencher
  - `premier`
  - `defiler`
- ❑ Nom de l'erreur : **"filePleine"**
- ❑ Condition de déclenchement  
Lorsque l'on tente d'ajouter un élément en fin = queue de file alors que celle-ci est pleine
- ❑ Opérations susceptibles de la déclencher
  - `enfiler`

## 7.- Code C++ d'un Gestionnaire de Files :

### Définition du type des éléments contenus dans la File

---

```
1  #ifndef FILE_H
2  #define FILE_H
3
4  #include <queue>
5  using namespace std;
6
7  // DEFINITION DU TYPE DES ELEMENTS CONTENUS DANS LA FILE
8
9  /* Pour faire une file avec des types de base :
10     typedef int UnElement; */
11
12  /* Pour faire une file avec des elements de type struct :
13     typedef struct
14     {
15         int coordX; // abscisse du point
16         int coordY; // ordonnee du point
17     } UnElement; */
18
19  /* Pour faire une file avec des elements dont le type est defini dans un autre
20     fichier :
21     #include "leFichierOuEstDefiniLeType.h"
22     typedef leTypeDefiniDansLeFichier UnElement; */
23
24  typedef int UnElement;
25
26  // DEFINITION DE LA FILE
27  typedef queue<UnElement> UneFile;
```

## 7.- Code C++ d'un Gestionnaire de Files : Primitives disponibles

---

```
29 // PRIMITIVES
30
31 void initialiser (UneFile& f);
32 // Initialise ou ré-initialise une file vide prête à l'emploi
33
34 unsigned int taille (const UneFile& f);
35 // Retourne le nombre d'elements stockes dans la file f
36
37 bool estVide (const UneFile& f);
38 // Retourne vrai si la file f est vide, faux sinon
39
40 UnElement premier (const UneFile& f);
41 /* Retourne l'element situe en tete de la file f
42    Genere l'exception "fileVide" si la file f est vide */
43
44 void enfiler (UneFile& f, UnElement e);
45 // Ajoute l'element e en queue de la file f
46
47 void defiler (UneFile& f);
48 /* Retire l'element situe en tete de la file f
49    Genere l'exception "fileVide" si la file f est vide */
50
51 void defiler (UneFile& f, UnElement& e);
52 /* Retire l'element situe en tete de la file f et le stocke dans e
53    Genere l'exception "fileVide" si la file f est vide */
54
55 #endif
```

## 8.- Utilisation d'une File :

### Exemple de code C++ utilisant une file d'entiers

```
1  #include <iostream>
2  using namespace std;
3  #include "file.h"
4
5  int main()
6  {
7      //Déclaration et initialisation de la file
8      UneFile maFile;
9      initialiser(maFile);
10
11     // Ajout d'éléments dans la file
12     enfiler(maFile, 0); enfiler(maFile, 1);
13     enfiler(maFile, 2); enfiler(maFile, 3);
14     enfiler(maFile, 4); enfiler(maFile, 5);
15
16     // Affichage du contenu de la file
17     cout << premier(maFile) << "... " ; defiler(maFile);
18     cout << premier(maFile) << "... " ; defiler(maFile);
19     cout << premier(maFile) << "... " ; defiler(maFile);
20     cout << premier(maFile) << "... " ; defiler(maFile);
21     cout << premier(maFile) << "... " ; defiler(maFile);
22     cout << premier(maFile) << "... " ; defiler(maFile);
23
24     return 0;
25 }
```

□ Affichage produit :

□ En ligne 23 (mode Debug), que valent les expressions

`estVide(maFile)?`

`estPleine(maFile)?`

`taille(maFile)?`

## 8.- Utilisation d'une File :

### Exemple de code C++ utilisant une file d'entiers

```
1  #include <iostream>
2  using namespace std;
3  #include "file.h"
4
5  int main()
6  {
7      //Déclaration et initialisation de la file
8      UneFile maFile;
9      initialiser(maFile);
10
11     // Ajout d'éléments dans la file
12     enfiler(maFile, 0); enfiler(maFile, 1);
13     enfiler(maFile, 2); enfiler(maFile, 3);
14     enfiler(maFile, 4); enfiler(maFile, 5);
15
16     // Affichage du contenu de la file
17     cout << premier(maFile) << "... " ; defiler(maFile);
18     cout << premier(maFile) << "... " ; defiler(maFile);
19     cout << premier(maFile) << "... " ; defiler(maFile);
20     cout << premier(maFile) << "... " ; defiler(maFile);
21     cout << premier(maFile) << "... " ; defiler(maFile);
22     cout << premier(maFile) << "... " ; defiler(maFile);
23
24     return 0;
25 }
```

□ Affichage produit : 0 ... 1 ... 2 ... 3 ... 4 ... 5 ...

□ En ligne 23 (mode Debug), que valent les expressions

`estVide(maFile)?` **TRUE**

`estPleine(maFile)?` **FALSE**

`taille(maFile)?` **0**

## 8.- Utilisation d'une File :

### Exemple de code C++ utilisant une file d'entiers

- But du code : afficher **tout** le contenu de la file

```
16 // Affichage du contenu de la file
17 cout << premier(maFile) << "... " ; defiler(maFile);
18 cout << premier(maFile) << "... " ; defiler(maFile);
19 cout << premier(maFile) << "... " ; defiler(maFile);
20 cout << premier(maFile) << "... " ; defiler(maFile);
21 cout << premier(maFile) << "... " ; defiler(maFile);
22 cout << premier(maFile) << "... " ; defiler(maFile);
```

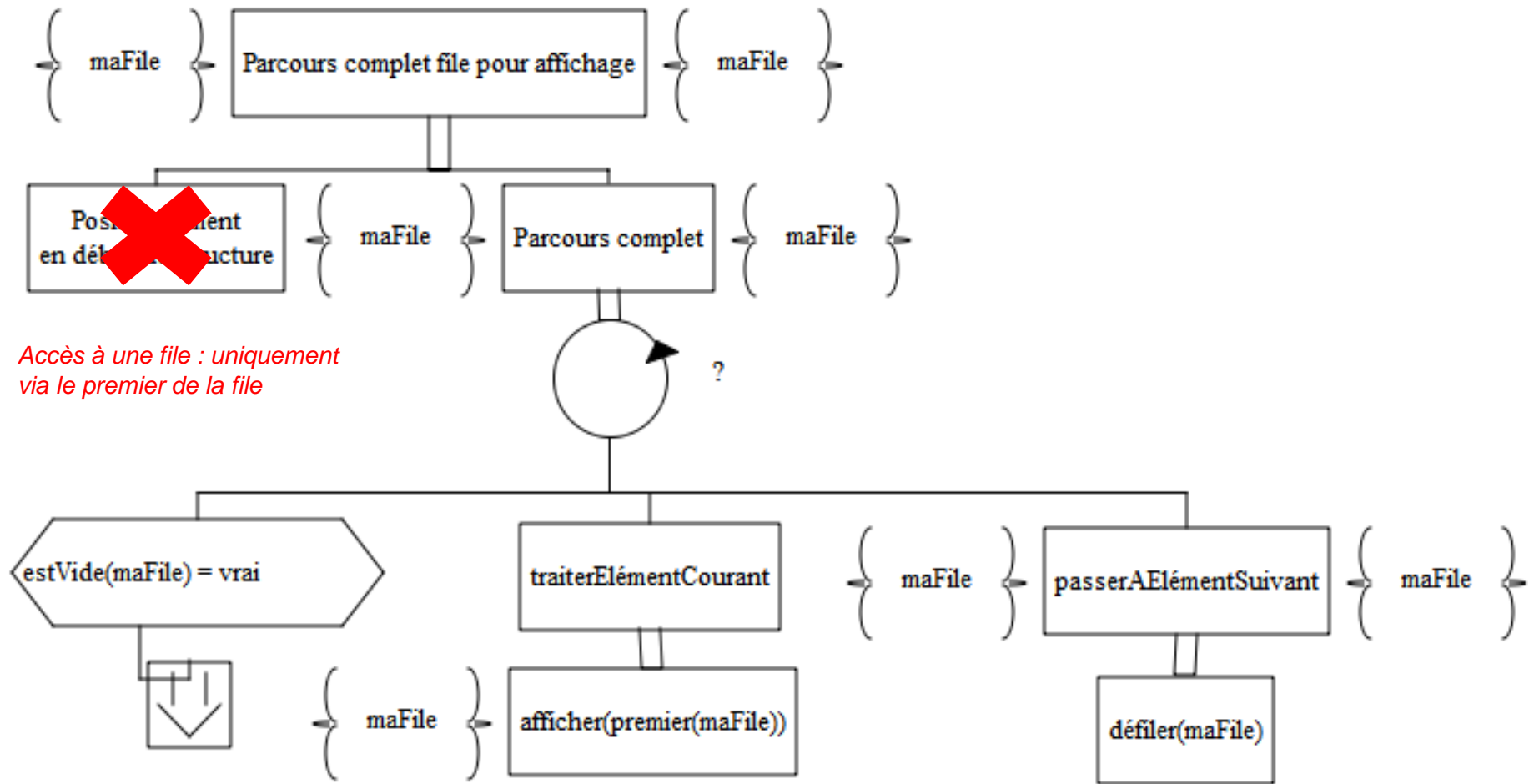
Modèle d'algorithme

= Parcours complet de la file avec traitement systématique  
Traitement à répéter : afficher élément courant

- Une file est une structure de données à accès séquentiel !  
→ voir Chapitre 2\_ModèlesAlgorithmesClassiques  
§3.- Parcours séquentiel complet avec traitement systématique

## 8.- Utilisation d'une File :

### Modèle d'algorithme associé



## 8.- Utilisation d'une File :

### Exemple de code C++ utilisant une file d'entiers

```
1  #include<iostream>
2  using namespace std;
3  #include "file.h"
4
5  int main(void)
6  {
7      // Déclaration et initialisation de la file
8      UneFile maFile;
9      initialiser(maFile);
10
11     // Ajout d'éléments dans la file
12     enfiler (maFile, 0); enfiler (maFile, 1);
13     enfiler (maFile, 2); enfiler (maFile, 3);
14     enfiler (maFile, 4); enfiler (maFile, 5);
15
16     // Affichage du contenu de la file
17     while ( !estVide(maFile) )
18     {
19         cout << premier(maFile) << "... ";
20         defiler(maFile);
21     }
22
23     return 0;
24 }
```

□ Affichage produit : 0 ... 1 ... 2 ... 3 ... 4 ... 5 ...

□ En ligne 22 (mode Debug), que valent les expressions

`estVide(maFile)?` **TRUE**

`estPleine(maFile)?` **FALSE**

`taille(maFile)?` **0**



chapitre 4.-  
Utilisation de *Piles* et de *Files*



Ressource R1.01 : Initiation au développement - Partie 2

**Merci pour votre attention !**