UNIVERSITÉ PARIS DESCARTES

SDA - Structures de Données et Algorithmes

T05 – Définition abstraite du TAD de File
T07 – Primitives de File
T20 à T21 – Définition abstraite du TAD de Liste
T23 – Primitives de Liste
Codage d'un composant de File
  T34 à T36 – Entête
  T37 à T39 – Corps
  T40 à T42 – Test
Codage d'un composant de Liste
  T43 à T45 – Entête
  T46 à T48 – Corps
  T49 à T51 – Test

Equipe pédagogique
Marie-José Caraty, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Eloi Keita

Cours n° 5

Type abstrait de données (TAD)
La file - définition abstraite et implémentation
La liste - définition abstraite et implémentation

Bibliographie

B. Stroustrup, Le langage C++, Campus Press.
V. Granet, Algorithmique et programmation en Java, Dunod.
C. Carrez, Structures de données en Java, C++ et Ada, Dunod.

Sommaire

RAPPEL

Définition d'un Type Abstrait de Données (TAD)

Définition d'un type abstrait de données
- une déclaration des TAD définis et utilisés ainsi que des constantes utilisées
- une description fonctionnelle des opérations
  (travaillant sur la donnée du TAD)
- une description axiomatique de la sémantique des opérations
  (support de preuve, démonstration formelle)

Définition du TAD **indépendante** de son **implémentation**
- des données (représentation)
  dite structure de données ou donnée « concrète »
- des opérations associées

Implémentation du TAD

Choix d'une donnée concrète pour stocker la donnée du TAD
dans un langage donnée (C++, Java, …)

Critères de la qualité de l'implémentation
- simplicité
- efficacité de l'accès aux données

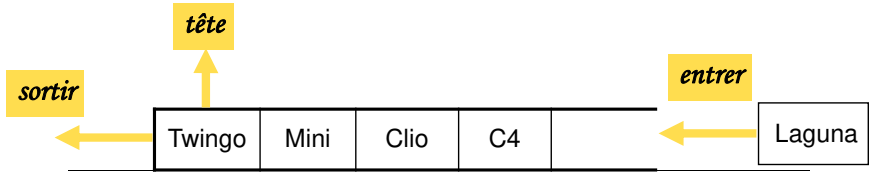
1. TYPE ABSTRAIT DE DONNEES de FILE

Introduction au TAD de file

Une file est une séquence (successeur défini sauf sur le dernier élément)
d'éléments accessibles par une seule extrémité appelée tête

Accès séquentiel à un élément,
sans exploitation de relation d'ordre entre éléments

Modèle **FIFO** (First In, First Out) premier entré, premier sort
Modèle type des **files d'attente**



Une file de quatre voitures en attente à un feu rouge

Définition abstraite du TAD de file

queue

Type abstrait de données : File
File **utilise** E (l'ensemble des éléments de file) et Booléen
fileVide ∈ File

Description fonctionnelle des opérations

push pop top

entrer :	File × E → File	Ajout d'un élément en queue de file
sortir :	File → File	Suppression de l'élément en tête de file
tête :	File → E	Lecture du premier élément de la file
estVide :	File → Booléen	Est-ce que la file est vide ?

Description axiomatique de la file

∀f ∈ File, ∀ e ∈ E

1) estVide(fileVide) = vrai
2) estVide(entrer(f, e)) = faux
3) f = fileVide, tête(entrer(f, e)) = e
4) sortir(entrer(f, e)) = f = entrer(sortir(f), e)
5) (⌉∃)f / f = sortir(fileVide)
6) (⌉∃)e / e = tête(fileVide)

Préconditions des opérations
∀f ∈ File, ∀ e ∈ E
sortir(f) : ⌉estVide(f)
tête(f) : ⌉estvide(f)

Manipulation du TAD de file

Déclarations d'utilisation de la file `fVoitures`

fVoitures ∈ File avec E: ensemble des modèles de voitures
fVoitures ← fileVide
e ∈ E

Suite d'actions

1) fVoitures ← entrer(fVoitures, Twingo)
2) fVoitures ← entrer(fVoitures, C4)
3) fVoitures ← entrer(fVoitures, Clio)
4) fVoitures ← entrer(fVoitures, Cooper)

Twingo	C4	Clio	Cooper	

5) fVoitures ← sortir(fVoitures)
6) fVoitures ← entrer (fVoitures, Laguna)
7) fVoitures ← sortir(fVoitures)
8) fVoitures ← sortir(fVoitures)
9) e ← tête(fVoitures)

Question 1) Quelle est la valeur de l'élément e ?
Question 2) Dessiner la file fVoitures

cf. Annexe du programme documenté

Spécification du composant (de) File – File.h

Donnée concrète de File

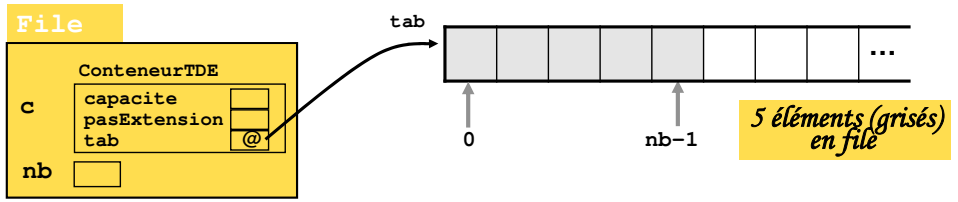
Opérations du composant File

```
#ifndef _FILE_
#define _FILE_

/**
 * @file File.h
 *
 * @brief Composant de file à capacité paramétrée
 */

#include "Item.hpp"
struct File {
   ...
   ...
   ...
   ...
};

void initialiser(File& f, ...);
...
...
bool estVide(const File& f);
void entrer(File& f, const Item& it);
Item tete(const File& f);
void sortir(File& f);
#endif
```

Opérations liées à l'implémentation du TAD Pile

Opérations du TAD Pile

1) fondée sur un tableau linéaire (extensible) (1/2)



Donnée concrète : Conteneur en mémoire dynamique à capacité extensible

Sémantique de la représentation

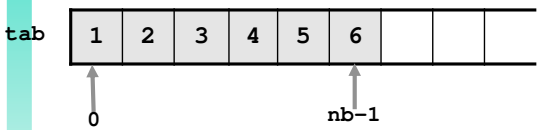
- `c` conteneur des éléments de type `ConteneurTDE` (en mémoire dynamique à capacité extensible)
- `nb` nombre d'éléments stockés dans la file

Objectif dans le codage de chacune des opérations

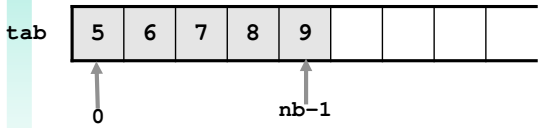
- gérer les attributs (`c` et `nb`) de la donnée concrète de file suivant la fonctionnalité de l'opération
- en cohérence avec les préconditions décrites en commentaire à contrôler dans le code par assertion

1) fondée sur un tableau linéaire (extensible) (2/2)

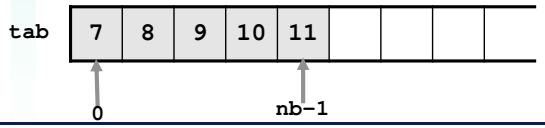
Entrée des éléments 1, 2, 3, 4, 5, 6



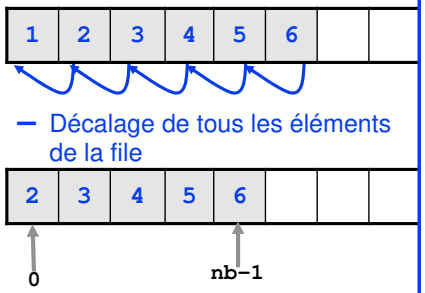
Sortie de 4 éléments et entrée de 7, 8, 9



Sortie de 2 éléments et entrée de 10, 11



Sortie d'un élément de file

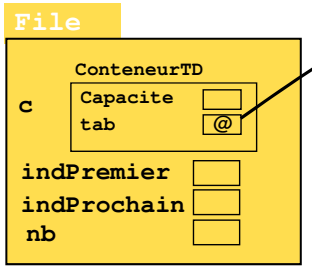


– Décalage de tous les éléments de la file

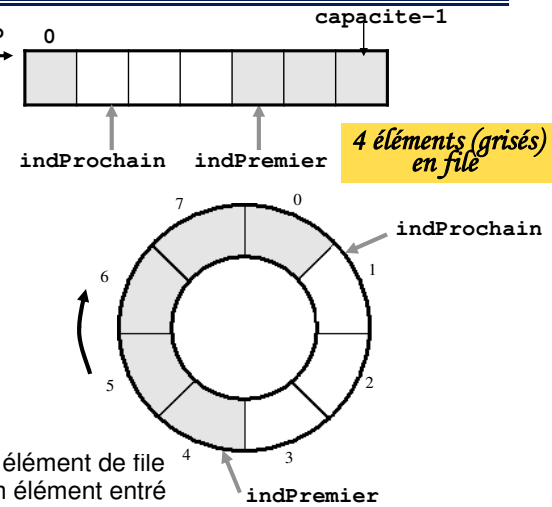
Coût (en temps d'exécution) de l'opération de sortie lié au choix de la **donnée concrète**

En grisé, les éléments en file

2) fondée sur un tableau circulaire (capacité limitée) (1/2)



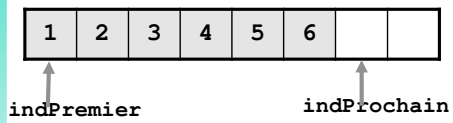
Donnée concrète de File
Tableau circulaire à capacité paramétrée



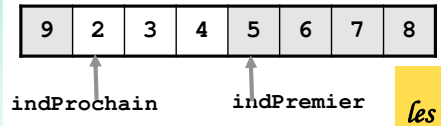
Sémantique de la représentation
- `c` conteneur des éléments
- `indPremier` indice du premier élément de file
- `indProchain` indice du prochain élément entré
- `nb` nombre d'éléments stockés dans la file

2) fondée sur un tableau circulaire (capacité limitée) (2/2)

Ajout des éléments 1, 2, 3, 4, 5, 6

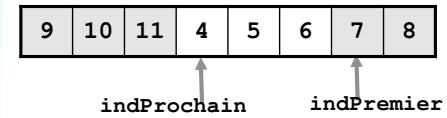


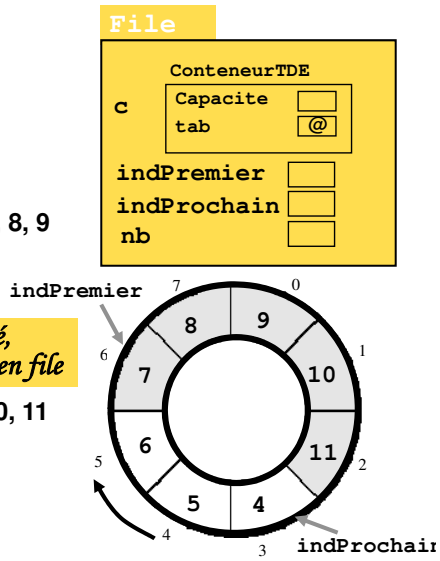
Retrait de 4 éléments et ajout des élément 7, 8, 9



En grisé, les éléments en file

Retrait de 2 éléments et ajout des élément 10, 11





cf. Annexe du programme documenté

Spécification du composant (de) File – File.h

```
#ifndef _FILE_
#define _FILE_

/**
 * @file File.h
 * @brief Composant de file circulaire à capacité paramétrée
 */

#include « ConteneurTD.h"
struct File {
 ConteneurTD c;            // conteneur des éléments de file
 unsigned int indPremier; // index de la tête de file dans tab
 unsigned int indProchain;// index du prochain élément entré en file
 unsigned int nb;          // nombre d'éléments dans la file
};

void initialiser(File& f, unsigned int capa);
void detruire(File& f);
bool estPleine(const File& f);
bool estVide(const File& f);
void entrer(File& f, Item it);
Item tete(const File& f);
void sortir(File& f);

#endif
```

Donnée concrète de File

Opérations du composant File

Opérations liées à l'implémentation du TAD Pile

Opérations du TAD Pile

Corps du composant (de) File – File.cpp (1/5)

Simulation de la constante fileVide du TAD

```
/**
 * @brief initialiser une file vide
 * la file est allouée en mémoire dynamique
 * @see detruire, elle est à désallouer en fin d'utilisation
 * @param[out] f : la file à initialiser
 * @param[in] capa : capacité de la file (nb maximum d'items stockés)
 * @pre capa>0
 */
```

Algorithme // Initialiser les attributs (c, indPremier, indProchain, nb) de f
// au valeurs d'une file vide de capacité capa
initialiser (à vide) f.c (dont l'allocation en mémoire dynamique)
f.indPremier ← f.indProchain ← f.nb ← 0

```
void initialiser(File& f, unsigned int capa) {
   assert(capa>0);
   initialiser(f.c, capa); // @see initialiser de ConteneurTD
   f.indPremier = 0;
   f.indProchain = 0;
   f.nb = 0;
}
```

Cas d'utilisation d'un conteneur de type ConteneurTD

Corps du composant (de) File – File.cpp (2/5)

```
/**
 * @brief Désallouer une file
 * @see initialiser, la file a déjà été allouée en mémoire dynamique
 * @param[out] f : la file
 */
```

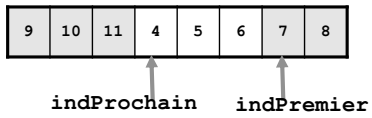
Algorithme Detruire f.c (désallocation de la mémoire dynamique)

```
void detruire(File& f) {
    detruire(f.c); // @see detruire de ConteneurTD
}
```

```
/**
 * @brief Lire l'item en tête de file
 * @param[in] f : la file
 * @return la valeur de l'item en tête de file
 * @pre la file n'est pas vide
 */
```

Algorithme Retourner l'élément d'index f.indPremier de f.c

```
Item tete(const File& f) {
   assert(!estVide(f));
   // @see lire, dans ConteneurTD
   return lire(f.c, f.indPremier);
}
```

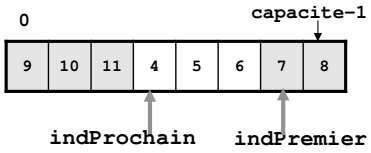


Corps du composant (de) File – File.cpp (3/5)

```
/**
 * @brief Test de file vide
 * @param[in] f : la file testée
 * @return true si f est vide, false sinon
 */
```

Algorithme f est vide si le nombre d'éléments est nul

```
bool estVide(const File& f) {
   return (f.nb == 0);
}
```




```
/**
 * @brief Test de file pleine
 * @param[in] f : la file testée
 * @return true si f est pleine, false sinon
 */
```

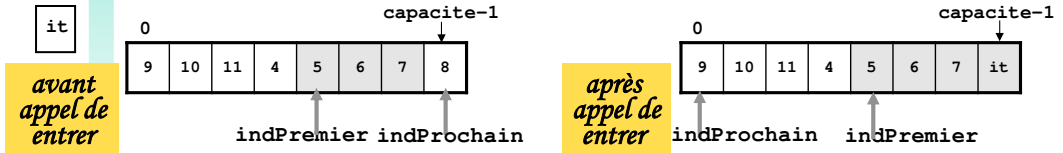
Algorithme f est pleine si le nombre d'éléments vaut la capacité du conteneur (f.c) de la file

```
bool estPleine(const File& f) {
   return (f.nb == f.c.capacite); // @see type ConteneurTD
}
```


Corps du composant (de) File – File.cpp (4/5)

```
/**
 * @brief Entrer un item dans la file
 * @param[in,out] f : la file
 * @param[in] it : l'item à entrer
 * @pre f n'est pas pleine
 */
```

Algorithme mémoriser it à l'indice f.indProchain dans f.c et mettre à jour f.nb et f.indProchain, l'indice de mémorisation du prochain item (incrémentation de 1 modulo f.c.capacite)

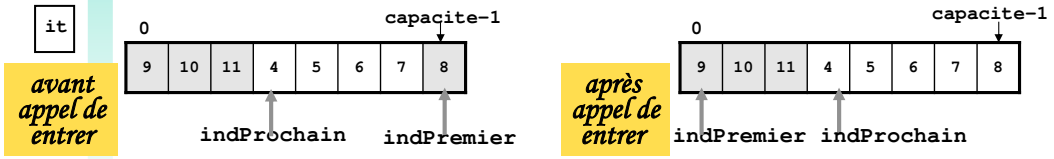



```
void entrer(File& f, const Item& it) {
   assert(!estPleine(f));
   ecrire(f.c, f.indProchain, it); //@see ecrire de ConteneurTD
   f.indProchain = (f.indProchain+1)%f.c.capacite;
   f.nb++;
}
```

Corps du composant (de) File – File.cpp (5/5)

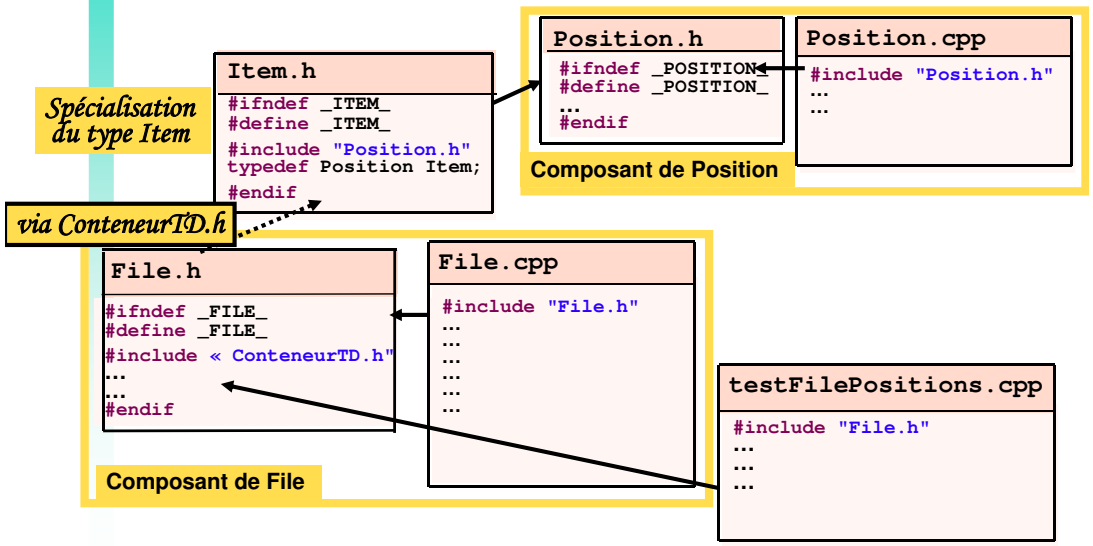
```
/**
 * @brief Sortir l'item tête de file
 * @param[in,out] f : la file
 * @pre f n'est pas vide
 */
```

Algorithme mettre à jour f.nb, le nombre d'éléments en file et f.indProchain, l'indice de mémorisation du prochain item (décrémentation de 1 modulo f.c.capacite)




```
void sortir(File& f) {
   assert(!estVide(f));
   f.indPremier = (f.indPremier+1)%f.c.capacite;;
   f.nb--;
}
```

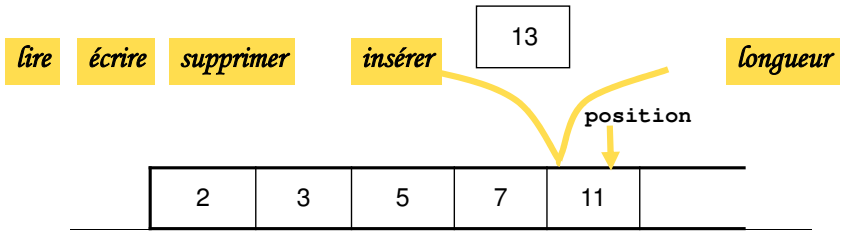

Graphe de dépendance des fichiers



Introduction au TAD de liste

Une liste est une séquence (successeur défini sauf sur le dernier élément) **non ordonnée dont les éléments appartiennent à un ensemble quelconque**

Accès direct (par position) à un élément,
sans exploitation de relation d'ordre entre éléments



Une liste de cinq nombres

Définition abstraite du TAD de Liste (1/2)

Type abstrait de données : Liste

Liste **utilise** E (ensemble des éléments de Liste), N (ensemble des entiers naturels) et Booléen

listeVide ∈ Liste

Description fonctionnelle des opérations

longueur :	Liste → N	Nombre d'éléments de la liste
lire :	Liste x N → E	Lecture de l'élément à la position donnée
écrire :	Liste x N x E → Liste	Ecriture d'un élément à une position donnée
insérer :	Liste x N x E → Liste	Insertion d'un élément à une position donnée
supprimer :	Liste x N → Liste	Suppression de l'élément à la position donnée

Définition abstraite du TAD de Liste (2/2)

Description axiomatique de la liste

∀l ∈ Liste, ∀ e ∈ E, ∀ (i, j) ∈ I

1) longueur(listeVide) = 0
2) longueur(écrire(l, i, e)) = longueur(l)
3) longueur(insérer(l, i, e)) = longueur(l)+1
4) longueur(supprimer(l, i, e) = longueur(l)-1
5) lire(écrire(l, i, e), i) = e
6) lire(insérer(l, i, e), j) ← si (j < i) alors lire(l, j) sinon lire(l, j+1)
7) lire(supprimer(l, i), j) ← si (j < i) alors lire(l, j) sinon lire(l, j-1)
8) supprimer(insérer(l, i, e), i) ← l
9) lire(l, i) : (0 ≤ i < longueur(l))
10) écrire(l, i) : (0 ≤ i < longueur(l))
11) supprimer(l, i) : (0 ≤ i < longueur(l))
12) insérer(l, i, e) : (0 ≤ i ≤ longueur(l))

Préconditions des opérations

Manipulation de la liste

Déclarations d'utilisation de la liste `lNombres`

lNombres ∈ Liste avec E: ensemble entiers positifs
lNombres ← listeVide

Suite d'actions

1) lNombres ← insérer(lNombres , 0, 2)
2) lNombres ← insérer(lNombres , 0, 3)
3) lNombres ← insérer(lNombres , 0, 5)
4) lNombres ← insérer(lNombres , 0, 7)
5) lNombres ← insérer(lNombres , 0, 11)

11	7	5	3	2	

6) lNombres ← insérer(lNombres, 2, 13)
7) lNombres ← supprimer(lNombres, 3)
8) e ← lire(lNombres, 3)

Question 1) Quelle est la valeur i) de l'élément e et ii) de longueur(lNombres) ?
Question 2) Dessiner la liste lNombres

cf. Annexe du programme documenté

Spécification du composant (de) Liste – Liste.h

Donnée concrète de Liste

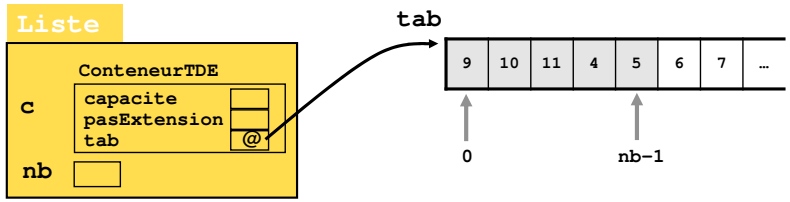
Opérations du composant Liste

```
#ifndef _LISTE_
#define _LISTE_

/**
 * @file Liste.h
 * ...
 * @brief Composant de liste
 */

#include "Item.h"
struct Liste {
   ...
   ...
   ...
};

void initialiser(Liste& l);
...
unsigned int longueur(const Liste& l);
Item lire(const Liste& l, unsigned int i);
void ecrire(Liste& l, unsigned int i, const Item& it);
void inserer(Liste& l, unsigned int pos, const Item& it);
void supprimer(Liste& l, unsigned int i);

#endif
```

Opérations liées à l'implémentation du TAD File

Opérations du TAD Liste

1) fondée sur un tableau dynamique (extensible)

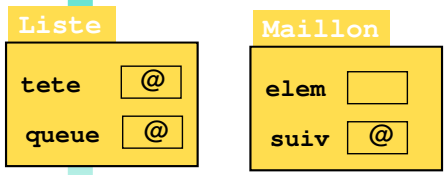


Donnée concrète :
Conteneur en mémoire dynamique à capacité extensible

Sémantique de la représentation

- **c**, conteneur des éléments de type `ConteneurTDE` (en mémoire dynamique à capacité extensible)
- **nb**, nombre d'éléments stockés dans la liste les éléments sont stockés de l'indice **0** à l'indice **nb-1**

2) fondée sur un chaînage simple (chaine de maillons)



Donnée concrète : liste simplement chaînée

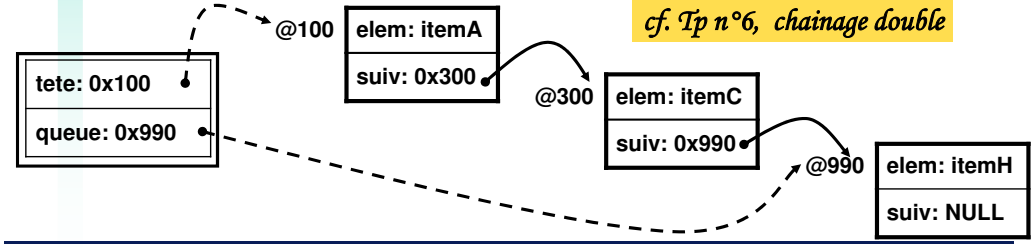
Sémantique de la représentation

- `tete`, pointeur sur le maillon de début de liste caractérise de manière unique une liste
- `queue`, pointeur sur le maillon de fin de liste (optimisation des opérations en fin de liste)

Type `Maillon`

- `elem`, élément de liste stocké dans un maillon
- `suiv`, pointeur sur le maillon suivant de la liste

cf. Tp n°6, chainage double



cf. Annexe du programme documenté

Spécification du composant (de) Liste – Liste.h

Donnée concrète de Liste

Opérations du composant Liste

Opérations liées à l'implémentation du TAD Liste

Opérations du TAD Liste

```
#ifndef _LISTE_
#define _LISTE_

/**
 * @file Liste.h
 * @brief Composant de liste fondé sur un conteneur extensible
 */

#include "ConteneurTDE.h"

struct Liste {
   ConteneurTDE c;// conteneur des éléments
   unsigned int nb;// nombre d'éléments stockés dans la liste
};

void initialiser(Liste& l, unsigned int capa, unsigned int pas);
void detruire(Liste& l);
unsigned int longueur(const Liste& l);
Item lire(const Liste& l, unsigned int i);
void ecrire(Liste& l, unsigned int pos, const Item& it);
void inserer(Liste& l, unsigned int pos, const Item& it);
void supprimer(Liste& l, unsigned int pos);

#endif
```


Corps du composant (de) Liste – Liste.cpp (1/5)

Simulation de la constante listeVide du TAD

```
/**
 * @brief Initialiser une liste vide
 * la liste est allouée en mémoire dynamique
 * @see detruire, la liste est à désallouer en fin d'utilisation
 * @param[out] l : la liste à initialiser
 * @param[in] capa : capacité de la liste
 * @param[in] pas : pas d'extension de la liste
 * @pre capa>0 et pas>0
 */
```

Algorithme // Initialiser les attributs (c et nb) de l au valeurs d'une liste vide
l.nb ← 0
l.c est un conteneur d'éléments de type Item alloué
en mémoire dynamique et extensible

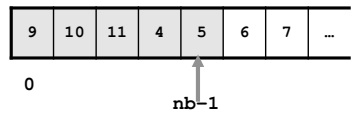
```
void initialiser(Liste& l, unsigned int capa, unsigned int pas) {
   assert ((capa>0) && (pas>0));
   initialiser(l.c, capa, pas);
   l.nb=0;
}
```


Corps du composant (de) Liste – Liste.cpp (2/5)

```
/**
 * @brief Désallouer une liste
 * @see initialiser, la liste a déjà été allouée en mémoire dynamique
 * @param[out] l : la liste
 */
```

Algorithme Désallouer l.tab en mémoire dynamique

```
void detruire(Liste& l) {
   detruire(l.c);
}
```




```
/**
 * @brief Longueur de liste
 * @param[in] l : la liste
 * @return la longueur de la liste
 */
```

Algorithme Retourner le nombre d'éléments de la liste

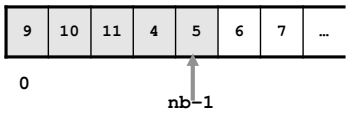
```
unsigned int longueur(const Liste& l) {
   return l.nb;
}
```

Corps du composant (de) Liste – Liste.cpp (3/5)

```
/**
 * @brief Lire un élément de liste
 * @param[in] l : la liste
 * @param[in] pos : position de l'élément à lire
 * @return l'item lu en position pos
 * @pre 0<=pos<longueur(l)
 */
```

Algorithme retourner l'élément stocké à la position pos dans l.c

```
Item lire(const Liste& l, unsigned int pos) {
   assert((pos>=0)&&(pos<l.nb));
   return lire(l.c, pos);
}
```




```
/**
 * @brief Ecrire un item dans la liste
 * @param[in,out] l : la liste
 * @param[in] pos : position de l'élément à écrire
 * @param[in] it : l'item
 * @pre 0<=pos<longueur(l)
*/
```

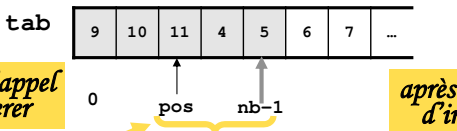
Algorithme écrire l'item it à la position pos dans l.c

```
void ecrire(Liste& l, unsigned int pos, const Item& it) {
   assert((pos>=0)&&(i<l.nb));
   ecrire(l.c, pos, it);
}
```

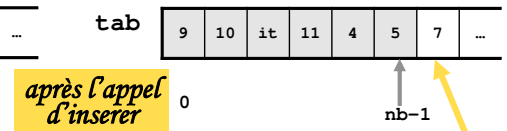

Corps du composant (de) Liste – Liste.cpp (4/5)

```
/**
 * @brief Insérer un élément dans une liste
 * @param[in,out] l : la liste
 * @param[in] pos : la position à laquelle l'élément est inséré
 * @param[in] it : l'élément inséré
 * @pre 0<=pos<=longueur(l)
 * l'insertion est faite avant la position donnée (pos)
 */
```

Algorithme **décaler** (d'une position vers la droite) **tous les éléments de la position pos à nb-1** écrire l'item it à la position pos, incrémenter de 1 la longueur de la liste



avant l'appel d'inserer



après l'appel d'inserer

Eléments à décaler (à droite)

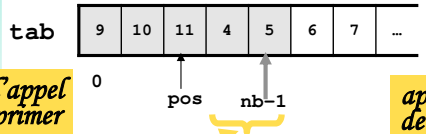
```
void inserer(Liste& l, unsigned int pos, const Item& it) {
   assert((pos>=0)&&(pos<=l.nb));
   for (int i=l.nb; i>pos; i--) {
      ecrire(l.c, i, lire(l.c, i-1));
   }
   ecrire(l.c, pos, it);
   l.nb++;
}
```

Rem: Insertion possible à la position nb

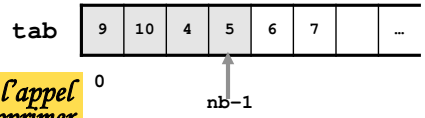
Corps du composant (de) Liste – Liste.cpp (5/5)

```
/**
 * @brief Supprimer un élément dans une liste
 * @param[in,out] l : la liste
 * @param[in] pos : la position de l'élément à supprimer
 * @pre longueur(l)>0 et 0<=pos<longueur(l)
 */
```

Algorithme **décaler** (d'une position vers la gauche) **tous les éléments de la position pos+1 à nb-1** décrémenter de 1 la longueur de la liste



avant l'appel de supprimer

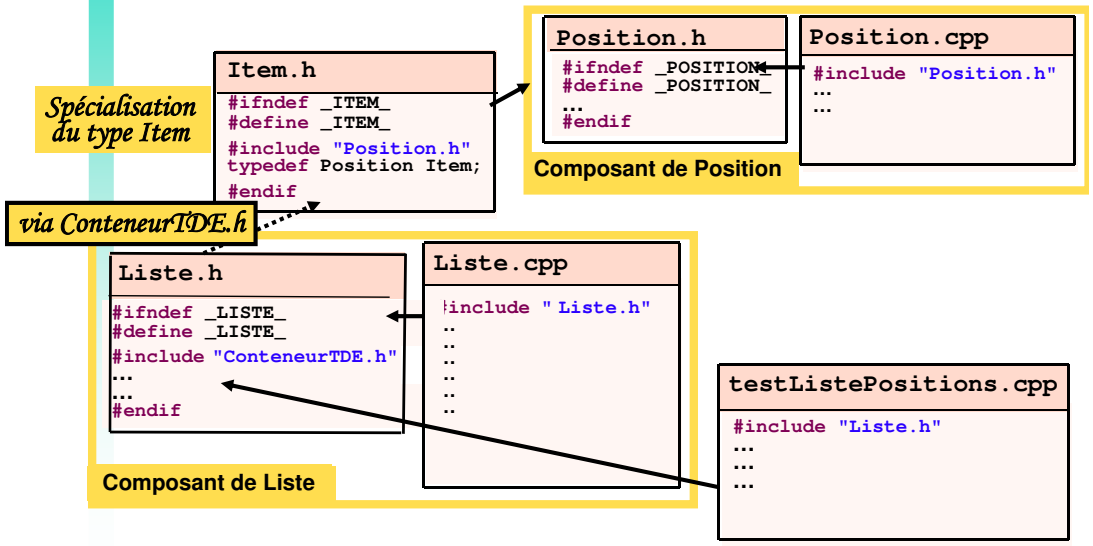


après l'appel de supprimer

Eléments à décaler (à gauche)

```
void supprimer(Liste& l, unsigned int pos) {
   assert((l.nb!=0) && (pos>=0) && (pos<l.nb));
   for (int i=pos; i<l.nb; ++i)
      ecrire(l.c, i, lire(l.c,i+1));
   l.nb--;
}
```


Graphe de dépendance des fichiers



Ce que vous avez appris aujourd'hui…

Deux autres TAD classiques en informatique : la **file** et la **liste**

Domaines d'utilisation de la file

- Gestion de ressources
  Modélisation de l'accès à une ressource (file d'attente)
- Planification
  Gestion de files en parallèle (gestion de stocks, péremption)
  Multiplexage de files (gestion de concurrences)

Domaines d'utilisation de la liste

- Manipulation (copie, échange, remplissage, remplacement, réorganisation, transformation)
- Recherche et filtrage (élément, motif)
- Ordonnancement (tri complet, tri partiel)

Le **problème posé** par le **choix de la donnée concrète** du TAD liée à **l'efficacité des opérations** (accès aux données)

Annexe du programme documenté

Spécification du composant (de) File – File.h (1/3)

sem06-cours-Cpp1

```
#ifndef _FILE_
#define _FILE_

/**
 * @file File.h
 * Projet sem06-cours-Cpp1
 * @author l'équipe pédagogique
 * @version 2 - 13/01/10
 * @brief Composant de file avec capacité paramétrée
 * Structures de données et algorithmes - DUT1 Paris 5
 */

#include « ConteneurTD.h"

struct File {
   ConteneurTD c; // conteneur des éléments de file
   unsigned int indPremier; // index de la tête de file dans tab
   unsigned int indProchain;// index du prochain élément entré en file
   unsigned int nb; // nombre d'éléments dans la file
};

/**
 * @brief initialiser une file vide
 * la file est allouée en mémoire dynamique
 * @see detruire, elle est à désallouer en fin d'utilisation
 * @param[out] f : la file à initialiser
 * @param[in] capa : capacité de la file (nb maximum d`items stockés)
 * @pre capa>0
 */
void initialiser(File& f, unsigned int capa);
```


Annexe du programme documenté

Spécification du composant (de) File – File.h (2/3)

```
/**
 * @brief Désallouer une file
 * @see initialiser, la file a déjà été allouée en mémoire dynamique
 * @param[out] f : la file
 */
void detruire(File& f);

/**
 * @brief Test de file pleine
 * @param[in] f : la file testée
 * @return true si f est pleine, false sinon
 */
bool estPleine(const File& f);

/**
 * @brief Test de file vide
 * @param[in] f : la file testée
 * @return true si f est vide, false sinon
 */
bool estVide(const File& f);

/**
 * @brief Lire l'item en tête de file
 * @param[in] f : la file
 * @return la valeur de l'item en tête de file
 * @pre la file n'est pas vide
 */
Item tete(const File& f);
```


Annexe du programme documenté

Spécification du composant (de) File – File.h (3/3)

```
/**
 * @brief Entrer un item dans la file
 * @param[in,out] f : la file
 * @param[in] it : l'item à entrer
 * @pre f n'est pas pleine
 */
void entrer(File& f, const Item& it);

/**
 * @brief Sortir l'item tête de file
 * @param[in,out] f : la file
 * @pre f n'est pas vide
 */
void sortir(File& f);

#endif
```

Annexe du programme documenté

Corps du composant (de) File – File.cpp (1/3)

sem06-cours-Cpp1

```
/**
 * @file File.cpp
 * Projet sem06-cours-Cpp1
 * @author l'équipe pédagogique
 * @version 2 - 13/01/10
 * @brief Composant de file avec capacité paramétrée
 * Structures de données et algorithmes - DUT1 Paris 5
 */

#include <cassert>
#include "File.h"

/**
 * @brief initialiser une file vide
 * la file est allouée en mémoire dynamique
 * @see detruire, elle est à désallouer en fin d'utilisation
 * @param[out] f : la file à initialiser
 * @param[in] capa : capacité de la file (nb maximum d'items stockés)
 * @pre capa>0
 */
void initialiser(File& f, unsigned int capa) {
   assert(capa>0);
   initialiser(f.c, capa); // @see initialiser de ConteneurTD
   f.indPremier = 0;
   f.indProchain = 0;
   f.nb = 0;
}
```


Annexe du programme documenté

Corps du composant (de) File – File.cpp (2/3)

```
/**
 * @brief Désallouer une file
 * @see initialiser, la file a déjà été allouée en mémoire dynamique
 * @param[out] f : la file
 */
void detruire(File& f) {
   detruire(f.c); // @see detruire de ConteneurTD
}

/**
 * @brief Test de file pleine
 * @param[in] f : la file testée
 * @return true si f est pleine, false sinon
 */
bool estPleine(const File& f) {
   return (f.nb == f.c.capacite); // @see type ConteneurTD
}

/**
 * @brief Test de file vide
 * @param[in] f : la file testée
 * @return true si f est vide, false sinon
 */
bool estVide(const File& f) {
   return (f.nb == 0);
}
```


Annexe du programme documenté

Corps du composant (de) File – File.cpp (3/3)

```
/**
 * @brief Lire l'item en tête de file
 * @param[in] f : la file
 * @return la valeur de l'item en tête de file
 * @pre la file n'est pas vide
 */
Item tete(const File& f) {
   assert(!estVide(f));
   // @see lire, dans ConteneurTD
   return lire(f.c, f.indPremier);
}

/**
 * @brief Entrer un item dans la file
 * @param[in,out] f : la file
 * @param[in] it : l'item à entrer
 * @pre f n'est pas pleine
 */
void entrer(File& f, const Item& it) {
   assert(!estPleine(f));
   ecrire(f.c, f.indProchain, it); //@see ecrire de ConteneurTD
   f.indProchain = (f.indProchain+1)%f.c.capacite;
   f.nb++;
}

/**
 * @brief Sortir l'item tête de file
 * @param[in,out] f : la file
 * @pre f n'est pas vide
 */
void sortir(File& f) {
   assert(!estVide(f));
   f.indPremier = (f.indPremier+1)%f.c.capacite;;
   f.nb--;
}
```


Annexe du programme documenté

Test du composant (de) File – testFilePositions.cpp (1/2)

sem06-cours-Cpp1

```
/**
 * @file testFilePositions.cpp
 * Projet sem06-cours-Cpp1
 * @author l'équipe pédagogique
 * @version 1 - 25/01/06
 * @brief Test du composant de file
 * Structures de données et algorithmes - DUT1 Paris 5
 */

#include <iostream>
using namespace std;

#include "File.h"

/* Test d'une file (en mémoire dynamique et à capacité paramétrée
 * d'éléments de type Position */
int main() {

   File fPositions; // Déclaration de la file de positions
   Position p;

   /* Initialisation de la file de capacité 5 */
   initialiser(fPositions, 5);

   cout << "Test de la file de positions" << endl;

   /* Test de file vide */
   cout << "Début de l'application : ";
   if (estVide(fPositions)) cout << "la file est vide" << endl;
   else cout << "la file n'est pas vide" << endl;
```

Test du composant (de) File – testFilePositions.cpp (2/2)

```
/* Ajout de positions dans la file jusqu'à la saisie
 * d'une date d'année nulle (date non empilée) */
cout << "Saisir des positions jusqu'à la saisie de l'origine [0,0]\n";
cout << "Les positions (à l'exception de celle de l'origine)\n";
cout << "seront ajoutées à la file de capacité "
     << fPositions.tab.capacite << endl;
bool estOrigine;
unsigned short nbSaisies =0;
do {
   p = saisir();
   estOrigine = (p.abscisse == 0) && (p.ordonnee == 0);
   if (!estOrigine) {
       nbSaisies++;
       entrer(fPositions, p);
   }
} while (!estOrigine);

cout << "Etat de la file après " << nbSaisies << " positions entrées : "
if (estVide(fPositions)) cout << "la file est vide" << endl;
else cout << "la file n'est pas vide" << endl;

/* Afficher l'état de la file (par défilage complet)  */
cout << "Défilage complet de la file : " << endl;
while (!estVide(fPositions)) {
   cout << "Elément sorti de file : ";
   afficher(tete(fPositions)); cout << endl; // affiche la tête de file
   sortir(fPositions); // sortie de l'élément en tête de file
}
cout << "Fin de défilage, la file est vide";

detruire(fPositions);
return 0;
}
```


Traces du test du composant (de) File – testFilePositions.cpp

```
Test de la file de positions
Début de l'application : la file est vide
Saisir des positions jusqu'à la saisie de l'origine [0,0]
Les positions (à l'exception de celle de l'origine)
seront ajoutées à la file de capacité 5
Position (abscisse? ordonnee?) ? 1 1
Position (abscisse? ordonnee?) ? 2 2
Position (abscisse? ordonnee?) ? 3 3
Position (abscisse? ordonnee?) ? 4 4
Position (abscisse? ordonnee?) ? 0 0
Etat de la file après 4 positions entrées : la file n'est pas vide
Défilage complet de la file :
Elément sorti de file : [1, 1]
Elément sorti de file : [2, 2]
Elément sorti de file : [3, 3]
Elément sorti de file : [4, 4]
Fin de défilage, la file est vide
```


Spécification du composant (de) Liste – Liste.h (1/3)

sem06-cours-Cpp2

```
#ifndef _LISTE_
#define _LISTE_

/**
 * @file Liste.h
 * Projet sem06-cours-Cpp2
 * @author l'équipe pédagogique
 * @version 2 - 13/01/10
 * @brief Composant de liste en mémoire dynamique et extensible
 * Structures de données et algorithmes - DUT1 Paris 5
 */

#include « ConteneurTDE.h"

struct Liste {
   ConteneurTDE c; // conteneur des éléments de la liste
   unsigned int nb;// nombre d'éléments stockés dans la liste
};

/**
 * @brief Initialiser une liste vide
 * la liste est allouée en mémoire dynamique
 * @see detruire, la liste est à désallouer en fin d'utilisation
 * @param[out] l : la liste à initialiser
 * @param[in] capa : capacité de la liste
 * @param[in] pas : pas d'extension de la liste
 * @pre capa>0 et pas>0
 */
void initialiser(Liste& l, unsigned int capa, unsigned int pas);
```


Spécification du composant (de) Liste – Liste.h (2/3)

```
/**
 * @brief Désallouer une liste
 * @see initialiser, la liste a déjà été allouée en mémoire dynamique
 * @param[out] l : la liste
 */
void detruire(Liste& l);

/**
 * @brief Longueur de liste
 * @param[in] l : la liste
 * @return la longueur de la liste
 */
unsigned int longueur(const Liste& l);

/**
 * @brief Lire un élément de liste
 * @param[in] l : la liste
 * @param[in] pos : position de l'élément à lire
 * @return l'item lu en position pos
 * @pre 0<=pos<longueur(l)
 */
Item lire(const Liste& l, unsigned int pos);
```

Annexe du programme documenté

Spécification du composant (de) Liste – Liste.h (3/3)

```
/**
 * @brief Ecrire un item dans la liste
 * @param[in,out] l : la liste
 * @param[in] pos : position de l'élément à écrire
 * @param[in] it : l'item
 * @pre 0<=pos<longueur(l)
*/
void ecrire(Liste& l, unsigned int pos, const Item& it);

/**
 * @brief Insérer un élément dans une liste
 * @param[in,out] l : la liste
 * @param[in] pos : la position à laquelle l'élément est inséré
 * @param[in] it : l'élément inséré
 * @pre 0<=pos<=longueur(l)
 * l'insertion est faite avant la position pos
 */
void inserer(Liste& l, unsigned int pos, const Item& it);

/**
 * @brief Supprimer un élément dans une liste
 * @param[in,out] l : la liste
 * @param[in] pos : la position de l'élément à supprimer
 * @pre longueur(l)>0 et 0<=pos<longueur(l)
 */
void supprimer(Liste& l, unsigned int pos);

#endif
```


Annexe du programme documenté

Corps du composant (de) Liste – Liste.cpp (1/3)

sem06-cours-Cpp2

```
/**
 * @file Liste.cpp
 * Projet sem06-cours-Cpp2
 * @author l'équipe pédagogique
 * @version 2 - 13/01/10
 * @brief Composant de liste en mémoire dynamique et extensible
 * Structures de données et algorithmes - DUT1 Paris 5
 */

#include <cassert>
#include "Liste.h"
/**
 * @brief Initialiser une liste vide
 * la liste est allouée en mémoire dynamique
 * @see detruire, la liste est à désallouer en fin d'utilisation
 * @param[out] l : la liste à initialiser
 * @param[in] capa : capacité de la liste
 * @param[in] pas : pas d'extension de la liste
 * @pre capa>0 et pas>0
 */
void initialiser(Liste& l, unsigned int capa, unsigned int pas) {
    assert ((capa>0) && (pas>0));
    initialiser(l.c, capa, pas);
    l.nb=0;
}
/**
 * @brief Désallouer une liste
 * @see initialiser, la liste a déjà été allouée en mémoire dynamique
 * @param[out] l : la liste
 */
void detruire(Liste& l) {
    detruire(l.c);
}
```


Annexe du programme documenté

Corps du composant (de) Liste – Liste.cpp (2/3)

```
/**
 * @brief Longueur de liste
 * @param[in] l : la liste
 * @return la longueur de la liste
 */
unsigned int longueur(const Liste& l) {
    return l.nb;
}

/**
 * @brief Lire un élément de liste
 * @param[in] l : la liste
 * @param[in] pos : position de l'élément à lire
 * @return l'item lu en position pos
 * @pre 0<=pos<longueur(l)
 */
Item lire(const Liste& l, unsigned int pos) {
    assert((pos>=0)&&(pos<l.nb));
    return lire(l.c, pos);
}

/**
 * @brief Ecrire un item dans la liste
 * @param[in,out] l : la liste
 * @param[in] pos : position de l'élément à écrire
 * @param[in] it : l'item
 * @pre 0<=pos<longueur(l)
*/
void ecrire(Liste& l, unsigned int pos, const Item& it) {
    assert((pos>=0)&&(pos<l.nb));
    ecrire(l.c, pos, it);
}
```


Annexe du programme documenté

Corps du composant (de) Liste – Liste.cpp (3/3)

```
/**
 * @brief Insérer un élément dans une liste
 * @param[in,out] l : la liste
 * @param[in] pos : la position à laquelle l'élément est inséré
 * @param[in] it : l'élément inséré
 * @pre 0<=pos<=longueur(l)
 * l'insertion est faite avant la position donnée (pos)
 */
void inserer(Liste& l, unsigned int pos, const Item& it) {
    assert((pos>=0)&&(pos<=l.nb));
    for (int i=l.nb; i>pos; i--) {
        ecrire(l.c, i, lire(l.c, i-1));
    }
    ecrire(l.c, pos, it);
    l.nb++;
}

/**
 * @brief Supprimer un élément dans une liste
 * @param[in,out] l : la liste
 * @param[in] pos : la position de l'élément à supprimer
 * @pre longueur(l)>0 et 0<=pos<longueur(l)
 */
void supprimer(Liste& l, unsigned int pos) {
    assert((l.nb!=0) && (pos>=0) && (pos<l.nb));
    for (int i=pos; i<l.nb; ++i)
        ecrire(l.c, i, lire(l.c,i+1));
    l.nb--;
}
```

Test du composant (de) Liste – testListePositions.cpp (1/2)

sem06-cours-Cpp2

```
/**
 * @file testListePositions.cpp
 * Projet sem06-tp-Cpp3
 * @author l'équipe pédagogique
 * @version 2 - 13/01/10
 * @brief Test d'une liste de positions
 * Structures de données et algorithmes - DUT1 Paris 5
 */

#include <iostream>
using namespace std;

#include "Liste.h"

/* Test d'une liste de positions */
int main(int argc, char* argv[]) {

   Liste lPositions; // Déclaration de la liste de positions
   Position p;

   initialiser(lPositions, 10,2);// Créer une liste de vide

   cout << "Test d'une liste de positions" << endl;

   /* Ajout de positions en début de liste
    * jusqu'à la saisie d'une position d'absisse 0 (non ajoutée) */
   cout << "Saisir des positions jusqu'à la saisie de l'origine [0,0]\n";
   cout << "Les positions (à l'exception de celle de l'origine)\n";
   cout << "seront ajoutées en début de liste" << endl;
   bool estOrigine;
```


Test du composant (de) Liste – testListePositions.cpp (2/2)

```
   do {
     p = saisir();
     estOrigine = (p.abscisse == 0) && (p.ordonnee == 0);
     if (!estOrigine) inserer(lPositions, 0, p);
   } while (!estOrigine);

   /* afficher tous les éléments de la liste */
   for (unsigned int i = 0; i < longueur(lPositions); ++i)
     afficher(lire(lPositions, i));
   cout << endl;

   cout << "Insertion de l'élément [7,8] en fin de liste\n";
   p.abscisse = 7; p.ordonnee = 8;
   inserer(lPositions, longueur(lPositions), p);

   /* suppression du deuxième élément de liste */
   cout << "Suppression du deuxième élément de liste\n";
   supprimer(lPositions, 1);

   cout << "Longueur de la liste :" << longueur(lPositions) << endl;

   cout << "Affichage de la liste :\n";
   for (unsigned int i = 0;i < longueur(lPositions);i++)
     afficher(lire(lPositions, i));
   cout << endl;

   /* destruction de la liste */
   detruire(lPositions);

   return 0;
}
```


Traces du test du composant (de) Liste – testListePositions.cpp

```
Test d'une liste de positions
Saisir des positions jusqu'à la saisie de l'origine [0,0]
Les positions (à l'exception de celle de l'origine)
seront ajoutées en début de liste
Position (abscisse? ordonnee?) ? 1 1
Position (abscisse? ordonnee?) ? 2 2
Position (abscisse? ordonnee?) ? 3 3
Position (abscisse? ordonnee?) ? 4 4
Position (abscisse? ordonnee?) ? 5 5
Position (abscisse? ordonnee?) ? 6 6
Position (abscisse? ordonnee?) ? 7 7
Position (abscisse? ordonnee?) ? 0 0
[7, 7] [6, 6] [5, 5] [4, 4] [3, 3] [2, 2] [1, 1]
Insertion de l'élément [7,8] en fin de liste
Suppression du deuxième élément de liste
Longueur de la liste :7
Affichage de la liste :
[7, 7] [5, 5] [4, 4] [3, 3] [2, 2] [1, 1] [7, 8]
```