UNIVERSITÉ PARIS DESCARTES

IUT

DÉPARTEMENT INFORMATIQUE

DISCIPLINE : SDA

Date de l'épreuve : 16/01/18

Année : 1 Groupe : 109

Écrire très lisiblement

NOM (en capitales) : APOSTOLET

Prénom : Arsène

NOTE DE 0 À 20 : 20/20 Beau ! Félicitations MK

APPRÉCIATIONS : I. (7.25) II (13) 1. 1.25 2. 1.25 3. 9 4. 1.5 — 20,5/20

Ne rien écrire dans cette marge

PARTIE 1 : QROC

1)

a)

bool foXou (Blob1 const &e1, int e2, Blob1 *& tab, unsigned int t, Blob2 & s);

0.5

b)

```
/**
*@file testfoXou.cpp
*@author Arsène Lapostolet
*@brief test de la fonction foXou
*@version 1 - 16/01/18
*/

#include "foXou.h"

int main(){
```

```
Blob1 a;
int b = 7;
unsigned int taille = 15;
Blob1 tableau[taille];
Blob2 c;
init(a);
for(unsigned int i = 0; i < taille; ++i){
    init(tableau[i]);
}
init( c );
// Appel foxou
foxou(a, b, tableau, t, c);
```

⚠ doit être une constante (et type entier)

0,75

TP>

2) Cette fonction ne vérifie pas la validité de ses paramètres par assertion. Cela n'est pas sécurisé. Par exemple, dans le cas de deux nombre positif >0 la condition d'arrêt n'est plus valable.

0.5

En effet, la valeur d'arrêt ($b == 0$) ne peut être atteinte pour $b > 0$ car à chaque appel de la fonction x, le deuxième paramètre (b) ~~est~~ croissant.

3) a) struct Point2D {
unsigned int abscisse;
unsigned int ordonnee;
};

0.25

struct Polygone {
unsigned int nbSommets;
~~Point2D sommets[nbSommets];~~
};

0.25

⚠ Point2D* sommets ;
(c'est le pointeur de l'allocation dynamique

b)

```
/**
 * @brief initialiser un polygone
 * @param[in-out] polygone à initialiser
 * @param[in] nb de sommets du polygone
 */
void initialiser(Polygone& p, unsigned int taille) {
    p.nbSommets = taille;
    p.sommets = new (nothrow) Point2D[p.nbSommets];
    if (t.sommets == NULL) {
        cout << "Allocation mémoire impossible";
        exit(1);
    }
    p.sommets = tSommets
}
```

quelques confusions

p.sommets

p.sommets

0.5

```
/**
 * @brief désallouer un polygone
 * @param[in-out] polygone à désallouer
```

*/
void desallouer(Polygone& p){
delete [] p.sommets;
} p.nbSommets = NULL;

0,25

b)

a) choix = 1
age = 23
poids = 63
nom = Marc

TB

0,75

b) choix = 9
age = 27
poids = 67,5
nom = Zoulou

TB

0,75

5) a) int e = tete(f);
b) sortir(f);
c) entrer(f, 92);
d) for(unsigned int i=0; i<5; ++i){
sortir(f);
}

0,5

mieux ! ?
while (!estVide(f))
sortir(f);

6) a) empiler(p, 57);
b) int e = sommet(p);
c) depiler(p);
d) for(unsigned int i=0; i<5; ++i){
depiler(p);
}

0,75

4,5)

7) a) inserer(l, 0, 57);
b) int e = lire(l, 8);

4

UNIVERSITÉ PARIS DESCARTES

IUT

DÉPARTEMENT INFORMATIQUE

DISCIPLINE : SDA

Date de l'épreuve : 16/01/18

Année : 1 Groupe : 109

Écrire très lisiblement

NOM : LAROSIÈRE (en capitales)

Prénom : Arsène

NOTE DE 0 À 20	APPRÉCIATIONS

Ne rien écrire dans cette marge

c) inserer(l, 3, 8);
d) supprimer(l, 8);
e) ecrire(l, 2, 6);
f) inserer(l, 6, -1);

Mieux : longueur(l)

0,75

8)

a) C'est le TAD de file qui est le plus adapté à la gestion d'un carrefour. En effet, il suit le modèle "First in, First out", ce qui suit bien la logique de la file d'attente des voitures à un carrefour équipé d'un feu de signalisation.

0.25

b) C'est le TAD de liste qui est le plus adapté. En effet, il permet de modifier n'importe quel élément mais aussi insérer des caractères n'importe où, ainsi que d'en supprimer. Cela est donc cohérent avec le cahier des charges du stockage du texte dans un logiciel de traitement de texte.

0.25

0,25

c) C'est le TAD de pile qui est le plus adapté. En effet, il permet d'accéder très rapidement au dernier élément entré et de le supprimer, ce qui est cohérent avec le fonctionnement de la fonction undo d'un traitement de texte.

PARTIE 2 : Gestionnaire de mémoire dynamique

1)

Q 1.1)

```
cout << "Blocs de memoire libres" << endl;
for(unsigned int i=0; i<longueur(bLibres); ++i){
    cout << "no : " << i << "  ad : " << lire(bLibres, i).adresse << "  t : " << lire(bLibres, i).taille << endl;
}
cout << "Blocs de memoire alloues" << endl;
for(unsigned int j=0; j<longueur(bAlloues); ++i){
    cout << "no : " << j << "  ad : " << lire(bAlloues, j).adresse << "  t : " << lire(bAlloues, j).taille << endl;
```

Nouvel écran dans l'exemple :

Blocs de mémoire libre
no:0 ad:8 t:19
Blocs de mémoire allouées
no:0 ad:0 t:13
no:1 ad:13 t:27
no:2 ad:40 t:33
no:3 ad:73 t:8

1.25

2)

Q2.1)

```
Bloc initialiser(unsigned int a, unsigned int t){
    Bloc b;
    b.adresse = a;
    b.taille = t;
    return b;
```

0.75

Q2.2)

```
ostream& afficher(ostream& os, const Bloc& b){
    for(unsigned int i=0; i<longueur(b.libres); ++i){
        if(b.a == lire(b.Libres,i).a){
            os << "no:" << i;
        }
    }
    for(unsigned int j=0; j<longueur(b.Alloues); ++i){
        if(b.a == lire(b.Alloues,i).a){
            os << "no:" << j;
        }
```

0.5

```
}
os << "ad : " << b.adresse << "t : " << b.taille <<
endl;
}
```

3)

3.1)

```
/**
 * @file gestionMemoire.h
 * @brief En tête du simulateur de mémoire
 * @author Arsène Lapostolet
 * @version 1 - 16/01/12
 **/
```

mais OK.

0.5

3.2)

```
/** @brief initialise le gestionnaire de mémoire
 * @param[in,out] g le gestionnaire
 * à initialiser
 * @param[in] tailleMemDyn la taille
 * de la mémoire dynamique
 * requise.
 * @pre tailleMemDyn > 0
 *
 */
void initialiser(GestionMemoire& g, unsigned
int tailleMemDyn) {
    assert(tailleMemDyn > 0);
    initialiser(g.bLibres, tailleMemDyn);
    initialiser(g.bAlloues, tailleMemDyn);
    Bloc b1 = initialiser(0, tailleMemDyn);
    .....
    inserer(g.bLibres, 0, b1);
}
```

0,5

2

7/8

UNIVERSITÉ PARIS DESCARTES

IUT

DÉPARTEMENT INFORMATIQUE

DISCIPLINE : S.D.A

Date de l'épreuve : 16/01/18

Année : 1 Groupe : 10

Écrire très lisiblement

NOM : (en capitales) LAPOSTOLET

Prénom : Alexis

NOTE DE 0 À 20	APPRÉCIATIONS

Ne rien écrire dans cette marge

3.3)

```
void detruire(GestionMemoire& g){
    detruire(g.blibres);
    detruire(g.bAlloues);
}
```

0.5 TB

3.4)

```
/**
 * @brief Alloue un bloc mémoire
 * @param[in-out] g gestionnaire de mémoire
 * @param[in] t taille du bloc à allouer
 * @return adresse en mémoire du
 * bloc alloué. -1 si impossible.
 */

int allouer(GestionMemoire& g, unsigned int t){
    for(unsigned int i=0; i<longueur(g.blibres);
    ++i){
        if(lire(g.blibres,i).taille >= t){
```

0.5

g.Livre::TableauMemDyn.taille - 1; i;
Bloc blocA = initialiser(t, lire(
bLibres, i).adresse);
inserer(g.bAlloues, (i), blocA);

return blocA.adresse;

esle {
return -1
}
}
}

3.5)

```
bool desallouer(GestionMemoire& m, unsigned
int a){
   for(unsigned int i=0; i<longueur(g.bAlloues);
   ++i){
      if (a == lire(bAlloues, i).adresse){
         supprimer(bAlloues, i);
         return 1;
      }
      else {
         return 0;
      }
   }
}
```

Il y a plus simple ! et plus lisible !

2

⚠ initialiser du bloc l'adresse d'abord !

⚠ pb d'accolades ?

1

+ insertion du bloc libre dans la liste triée !

3.6)

```
/**
 * @brief Affiche l'état de la mémoire
 * @param [in-out] g la mémoire à afficher
 * @return un flot de sortie
 */

ostream& afficher(ostream& os, const GestionMemoire& g) {
    os << "Blocs de mémoire libres" << endl;
    for (unsigned int i=0; i<longueur(g.bLibres); ++i) {
        os << "no : " << i << " ";
        afficher(os, lire(g.bLibres, i));
        os << endl;
    }
    os << "Blocs de mémoire allouées" << endl;
    for (unsigned int i=0; i<longueur(g.bAllouees); ++i) {
        os << "no : " << i << " ";
        afficher(os, lire(g.bAllouees, i));
        os << endl;
    }
}
```

4.1)

```
/**
 * @file testMemoire.cpp
 * @brief Programme de test du gestionnaire de mémoire
 * @author Alexia Laportolet
 * @version 1 - 16/04/18
 */
#include "GestionMemoire.h"
#include <iostream>
using namespace std;
```

```
int main() {
    GestionMemoire Gestionnaire;
    initialise(Gestionnaire, 100);
    b1=allouer(Gestionnaire, 13);
    allouer(Gestionnaire, 77);
    allouer(Gestionnaire, 33);
    allouer(Gestionnaire, 8);
    desallouer(Gestionnaire, Gestionnaire.libre
    (b(libres, 2).adresse)
    b3=allouer(20);
    desallouer(Gestionnaire, Gestionnaire.libre
    (b(libres, 3).adresse);
    allouer(Gestionnaire, 25);
    cout<<(Gestionnaire.b(libres[3].Adresse;

    Affiche(cout, Gestionnaire);
    system("pause");
    return 0;
}
```

4.2) Jeu d'essai:

20 TB

Blocs de mémoire libre

no: 0 ad: 65 t: 8

no: 1 ad: 73 t: 26

Blocs de mémoire allouée

no: 0 ad: 0 t: 13

no: 1 ad: 13 t: 27

no: 2 ad: 40 t: 25 TB

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année
SDA - Structures de Données et Algorithmes
Equipe pédagogique
Marie-José Caraty, Denis Poitrenaud,
Julien Rossit, Camille Kurtz,
Jacques Alès-Bianchetti, Eloi Keita

DST – 16 janvier 2018
STRUCTURES DE DONNEES ET ALGORITHMES

Durée : 3 heures
Tout document autorisé – Sans calculatrice

Les parties I et II sont indépendantes et les barèmes donnés sont indicatifs.
Toutes les spécifications données qui ne seront pas suivies seront pénalisées.
La documentation de code sera explicitement demandée dans la question.
La programmation demandée est impérative en Langage C++.
Prenez le temps de lire tout le sujet avant de commencer.

Partie I. QROC [8 points]

(1) Prototypage

L'interface de la fonction `forYou` est la suivante :
- deux paramètres d'entrée (e1 de type `Blob1`, `e2` de type entier naturel),
- un paramètre d'entrée-sortie (`tab`, tableau d'éléments de type `Blob1` de taille `t`),
- un paramètre de sortie (`s` de type `Blob2`)
- un paramètre de retour de type booléen

Spécification : les types `Blob1` et `Blob2` sont de grande taille-mémoire.

a) Suivant les règles de qualité du prototypage en C++, prototypez la fonction `forYou`.

b) Codez dans un programme principal un appel cohérent de la fonction `forYou`,

Spécification : vous utiliserez la fonction `init(Blob1& b)` qui initialise aléatoirement un paramètre b de type Blob1.,

(2) Récursivité

La fonction `r` à la page suivante n'est pas un bon exemple de fonction récursive. Donnez-en la raison.

```
int r(unsigned int a, unsigned int b) {
     if (b==0)
          return a;
     else
          return r(a+1, a+b);
}
```

(3) Allocation mémoire

Dans l'espace à 2 dimensions, un polygone est défini par ses `n` sommets (points `P(x, y)` de R^2). Dans une application, on choisit de stocker les sommets d'un polygone dans un tableau alloué en mémoire dynamique.

(a) Déclarez les types `Point2D` (représentant un point de N^2) et `Polygone` (représentant un polygone).

(b) Définissez (prototypez et codez) les fonctions `initialiser` et `desallouer` d'une variable `p` de type `Polygone`. Spécification : la fonction `initialiser` doit d'une part allouer en mémoire dynamique les conteneurs concernés et initialiser les autres attributs du type.

(4) Lecture de flot

Soit la lecture à partir du flot cin des variables suivantes :

```
char nom[20];
int age;
char choix;
float poids;
cin >> choix >> age >> poids >> nom;
```

Donner le contenu des variables `choix`, `age`, `poids` et `nom` dans des deux cas de saisie des données où <CR> symbolise le retour à la ligne (*Carriage Return*).

```
(a) 1 23 63        Marc Antoine<CR>
(b) q       27       67.5 2015 Lulla<CR>
```

(5) File

Soit la file `f` d'entiers contenant respectivement les éléments 6, 4, 7, 13, -28 et un entier `e`. La file est affichée avec le format suivant : [6, 4, 7 13, -28 [où 6 est la tête de file. Donnez la suite des instructions qui produiront respectivement les modifications de (a) à (d) visualisées ici par affichage. Spécification : Les instructions d'affichage ne sont pas demandées.

```
f <- [ 6, 4, 7, 13, -28 [

(a) e <- 6
(b) f <- [ 4, 7, 13, -28 [
(c) f <- [ 4, 7, 13, -28, 92 [
(d) estVide(f) <- true
```

(6) Pile

Soit la pile p d'entiers contenant respectivement les éléments 6, 4, 7, 13, -28 et un entier e. La pile est affichée avec le format suivant :] 6, 4, 7 13, -28] où 6 est le sommet de pile. Donnez la suite des instructions qui produiront respectivement les modifications de (a) à (d) visualisées ici par affichage. Spécification : Les instructions d'affichage ne sont pas demandées.

```
p <- ] 6, 4, 7 13, -28 ]

(a) p <-] 57, 6, 4, 7, 13, -28 ]
(b) e <- 57
(c) p <-] 6, 4, 7, 13, -28 ]
(d) estVide(p) <- true
```

(7) Liste

Soit la liste l d'entiers contenant respectivement les éléments 6, 4, 7, 13, -28 et un entier e. La liste est affichée avec le format suivant : { 6, 4, 7, 13, -28 }.
Donnez la suite des instructions qui produiront respectivement les modifications de (a) à (f) visualisées ici par affichage. Spécification : Les instructions d'affichage ne sont pas demandées.

```
l <- { 6, 4, 7, 13, -28}

(a)   l <- {57, 6, 4, 7, 13, -28}
(b)   e <- 13
(c)   l <- {57, 6, 4, 8, 7, 13, -28}
(d)   l <- {57, 6, 8, 7, 13, -28}
(e)   l <- {57, 6, 4, 7, 13, -28}
(f)   l <- {57, 6, 4, 7, 13, -28, -1}
```

(8) TAD

Parmi les Types Abstraits de Données (TAD) étudiés :

(a) Quel TAD est le plus adapté à la modélisation de la gestion du trafic à un carrefour équipé de feux de signalisation ?

(b) Dans un traitement de texte, on doit pouvoir dans l'ensemble du texte édité supprimer/ajouter/modifier un caractère, un mot, une phrase, un paragraphe, ... Quel TAD est le plus adapté au stockage du texte (sous forme de TAD de caractères, de TAD de mots, de TAD de phrase et de TAD de paragraphes ?

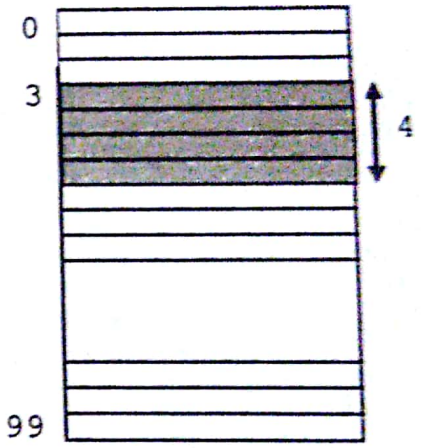
(c) La fonction undo d'un traitement de texte, permet d'annuler la/les dernière(s) commande(s). Quel TAD est utilisé pour traiter cette fonctionnalité ?

Partie II. Gestionnaire de la mémoire dynamique [13 points]

Les fonctions de la bibliothèque standard **malloc()**, **calloc()**, **realloc()**, **new**, **free()** et **delete** demandent de la mémoire au système ou libèrent de la mémoire. Cette partie vise la simulation de la gestion de la mémoire dynamique.

Le principe de gestion de la mémoire dynamique est de maintenir cohérentes deux listes : (1) la liste des blocs-mémoire libres et (2) la liste des blocs-mémoire alloués. Quand l'utilisateur demande l'allocation mémoire d'un bloc de taille **t**, un bloc libre de taille au moins égal à **t** est cherché dans la liste des blocs libres. Le bloc obtenu dépend de la stratégie utilisée. Parmi des stratégies classiques, on utilisera la stratégie *first fit* qui consiste à choisir le premier bloc libre de la liste tel que sa taille **t'** soit supérieure ou égale à **t**. La liste des blocs libres est maintenue par le gestionnaire de la mémoire par ordre croissant des adresses de blocs.

Dans la simulation de la gestion de la mémoire considérée, la mémoire dynamique disponible a une taille **tailleMemDyn** initialement fixée. L'unité de la mémoire dynamique adressable est l'octet, chaque octet est accessible par une adresse de **0** à (**tailleMemDyn-1**). Un bloc en mémoire est caractérisé (i) par son adresse d'allocation en mémoire dans l'intervalle [**0**, **MemDyn**[et (ii) par la taille du bloc exprimée en octet(s). Dans l'exemple ci-joint, la taille de la mémoire est 100 et le bloc grisé de taille 4 est alloué à l'adresse 3.



Les composants **Bloc** et **GestionMemoire** ont été développés pour la simulation de la gestion mémoire. Les entêtes des composants sont donnés (cf. Annexes 1 et 2, pages 6 et 7). Le composant **GestionMemoire** utilise le composant **Liste** (cf. Annexe 3, page 8) dont le type des éléments (**Item**) est spécialisé au type **Bloc**.

1. Algorithme de gestion mémoire

En considérant une mémoire dynamique de taille 100 et une stratégie de gestion du *first fit*, des allocations mémoire sont successivement demandées pour 4 blocs b1, b2, b3 et b4 respectivement de taille 13, 27, 33 et 8.

Q1.1. Donner l'état de la liste des blocs libres et de la liste des blocs alloués suivant le format de la trace de l'allocation mémoire à la page suivante. Dans le format, **no** indique le numéro de bloc dans la liste considérée (cadré à gauche sur 6 caractères), **ad** indique l'adresse du bloc en mémoire et **t**, sa taille (ad et t sont cadrés à gauche sur 4 caractères). cf. Format de la trace d'allocation page suivante.

```
Blocs de memoire libres
no:0       ad:… t:…
Blocs de memoire alloues
no:0       ad:…… t:……
no:1       ad:…… t:……
no:2       ad:…… t:……
```

Format de la trace de l'allocation mémoire

2. Composant Bloc

L'entête du composant se trouve en Annexe 1. L'objectif de la fonction initialiser est d'initialiser un bloc de données à partir de son adresse de début (a) et de sa taille (t).

Q2.1. Définissez la fonction initialiser.

Q2.2. Définissez la fonction afficher suivant le format précédemment donné à la question 1.1.

3. Composant GestionMemoire

Référez-vous au composant `GestionMemoire` (p. 7) pour chacune des questions.

Q3.1. Donnez la cartouche du gestionnaire de la mémoire (`GestionMemoire.h`).

Q3.2. L'objectif de la fonction **`initialiser`** est d'initialiser les attributs du gestionnaire de mémoire (champs de la structure **`GestionMemoire`**) : d'allouer et d'initialiser les listes vides (liste des blocs libres et liste des blocs alloués), puis de mettre à jour la liste des blocs libres avec l'unique bloc mémoire correspondant à la mémoire dynamique en fonction de la taille de la mémoire dynamique demandée.

Prototypez, documentez et codez la fonction **`initialiser`**.
Spécification : Précondition requise.

Q3.3. Codez la fonction **`detruire`** du gestionnaire de la mémoire.

Q3.4. L'objectif de la fonction **`allouer`** est de demander au gestionnaire mémoire l'allocation d'un bloc de mémoire de taille `t` et de récupérer l'adresse du bloc alloué en mémoire (`-1` en cas d'allocation mémoire impossible). La stratégie du *first fit* est utilisée et les listes des blocs libres et des blocs alloués sont à mettre à jour.

Prototypez, documentez et codez la fonction **`allouer`**.

Q3.5. Codez la fonction **`désallouer`**. Vous veillerez à ce que la liste des blocs libres soit maintenue dans l'ordre croissant des adresses des blocs libres.

Q3.6. Codez la fonction **`afficher`** qui affiche dans un flot de sortie l'état de la mémoire en affichant la liste des blocs libres et celle des blocs alloués (cf. `afficher` p. 6).

4. Programme de test

Q4.1. Codez le programme principal correspondant au scénario suivant : on considère une mémoire dynamique de taille 100 et une stratégie de gestion du *first fit*, demander successivement l'allocation de mémoire des 4 blocs b1, b2, b3 et b4 respectivement de taille 13, 27, 33 et 8. Désallouer le bloc b3, puis réallouer b3 de taille 40. Désallouer le bloc b4, puis réallouer b4 de taille 25. Afficher l'adresse de b4 et l'état de la mémoire.

Q4.2. Donnez la trace d'exécution attendue du programme.

Question Bonus

Les différentes allocations mémoire finissent par produire une fragmentation de la mémoire qui devient problématique pour les nouvelles allocations de blocs de grande taille. Le gestionnaire de la mémoire doit alors réorganiser l'occupation mémoire de manière à maximiser les blocs contigus de mémoire libre. Donnez une stratégie de réorganisation pour pallier la fragmentation mémoire. Prototypez et codez la fonction de réorganisation mémoire.

```
#ifndef BLOC_H_
#define BLOC_H_

struct Bloc {
   unsigned int adresse; // adresse du bloc de mémoire
   unsigned int taille;  // taille du bloc de mémoire
};

/**
 * @brief Initialiser un bloc de mémoire
 * @param[in] a : adresse du bloc de mémoire
 * @param[in] t : taille du bloc de mémoire
 * @return le bloc initialisé
 * @pre t>0
*/
Bloc initialiser(unsigned int a, unsigned int t);

/**
 * @brief Afficher un bloc-mémoire
 * @param[in,out] os : flot de sortie
 * @param[in] b : bloc-mémoire
 * @return os
*/
ostream& afficher(ostream& os, const Bloc& b);
#endif /*BLOC_H_*/
```

Annexe 1 – Entête du composant de bloc-mémoire

```
#ifndef GESTIONMEMOIRE_H_
#define GESTIONMEMOIRE_H_

/**
* Cartouche à compléter
**/

#include "Liste.h"

struct GestionMemoire {
    Liste bLibres;  // liste des blocs de mémoire libres
    Liste bAlloues; // liste des blocs de mémoire alloués
};

/**
 * Documentation à compléter
 */
_____ initialiser(_____);

/**
 * @brief Detruire/desallouer un gestionnaire de mémoire
 * @param[in,out] m : le gestionnaire de mémoire à désallouer
 */
void detruire(GestionMemoire& m);

/**
 * Documentation à compléter
 */
_____ allouer(_____);

/**
 * @brief Désallouer un bloc en mémoire
 * @param[in,out] m : le gestionnaire de mémoire
 * @param[in] a : l'adresse du début de bloc mémoire à désallouer
 * @return indicateur de désallocation
 *         vrai pour un bloc existant (dans la liste des blocs
 *         alloués) désalloué, faux sinon
 */
bool desallouer(GestionMemoire& m, unsigned int a);

/*
* Documentation à compléter
*/
_____ afficher(_____);

#endif /*GESTIONMEMOIRE_H_*/
```

Annexes 2 – Entête du composant de gestion mémoire

```
#ifndef LISTE_H
#define LISTE_H
/**
 * @file Liste.h
 * @brief Liste en mémoire dynamique à capacité paramétrée */
#include "TableauMemDyn.h"

struct Liste {
   TableauMemDyn tab; // tableau mémorisant les 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] c : capacité (>0) de la liste
 * @pre c>0 */
void initialiser(Liste& l, unsigned int c);
/**
 * @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);
/**
 * @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 /*LISTE_H_*/
```

Annexe 3 – Entête du composant de Liste