

► LIFAP3 : Algorithmique et programmation procédurale

Fascicule de travaux dirigés et de travaux pratiques

Nicolas Pronost, Samir Akkouche et Carole Knibbe ► Université Lyon 1

Notes :

Table des matières

Travaux dirigés.....	3
TD1 : Vie et mort des variables en mémoire (1/2)	4
TD2 : Vie et mort des variables en mémoire (2/2)	8
TD3 : Classes et objets	12
TD4 : Algorithmes, invariants de boucle et complexité.....	15
TD5 : Algorithmes sur les fichiers.....	21
TD6 : Exercices de synthèse.....	25
TD7 : Tableaux dynamiques	28
TD8 : Listes chaînées	30
TD9 : Piles et Files	32
TD10 : Arbres binaires	34
Travaux pratiques	35
TP1 : De LIFAP1 à LIFAP3... ..	36
TP2 : Vie et mort des variables en mémoire.....	41
TP3 : Classes et objets	43
TP4 : Fichiers, complexité et tris fusion.....	45
TP5 : Tableau dynamique.....	47
TP6 : Liste doublement chaînée.....	49
TP7 : Piles et Files.....	51
TP8 : Arbre binaire de recherche.....	52
Annexes.....	54
Annexe A : Commandes Linux usuelles	55
Annexe B : Tirage de nombres aléatoires	56
Annexe C : Mesure de temps d'exécution	57

Travaux dirigés

Notes :

[illegible]

TD1 : Vie et mort des variables en mémoire (1/2)

Exercice 1 : Appel à une fonction récursive

Donnez la trace mémoire et la trace écran du programme suivant. Pour la trace mémoire, utilisez le modèle de pile vu en cours. Vous supposerez que la valeur de retour du programme principal est stockée à l'adresse 3 965 281 684 et qu'un int occupe 4 octets. Vous dessinerez une « photo » de la pile à chaque fois qu'une frame est sur le point d'être supprimée. Vous ne dessinerez pas les frames des impressions cout.

```
#include <iostream>
using namespace std;

/* Calcul d'un coefficient binomial à l'aide du triangle de Pascal */
int comb(int n, int p) {
    int tmp1, tmp2;
    cout << "Calcul du nb de combinaisons de "<< p << " elts parmi "<< n << endl;

    if ((p==0) || (n==p))
        return 1;
    tmp1 = comb(n-1, p-1); /* premier appel récursif */
    tmp2 = comb(n-1, p);    /* second appel récursif */
    return tmp1 + tmp2;
}

int main() {
    int c;
    c = comb(4, 3);
    cout << "c vaut " << c << endl;
    return 0;
}
```

Exercice 2 : Appel à une procédure avec effet de bord

Soit le programme C++ suivant :

```
void proc(int & i, int j) {
    /* dessiner l'état de la mémoire */
    i = (i + j) * 2;
    j = j + 1;
    /* dessiner l'état de la mémoire */
}

int main() {
    int a=2, b=3;
    /* dessiner l'état de la mémoire */
    proc(a,b);
    /* dessiner l'état de la mémoire */
    return 0;
}
```

En utilisant le modèle de pile vu en cours, dessinez l'état de la mémoire à chaque fois que c'est indiqué en commentaire. Vous supposerez que les variables sont stockées à partir de l'adresse 0xbffff3a0 (note : bffff3a0 en base 16 = 3 221 222 304 en base 10).

Qu'en déduisez-vous au sujet du mode de passage de i lorsqu'il est passé en paramètre à *proc* ?

Exercice 3 : Appel à une fonction qui retourne une structure

Donnez la trace mémoire et la trace écran du programme suivant. Pour la trace mémoire, utilisez le modèle de pile vu en cours. Vous supposerez que la valeur de retour du programme principal est stockée à l'adresse 3 875 346 084 et qu'un int occupe 4 octets. Vous dessinerez une « photo » de la pile à chaque fois qu'une frame est sur le point d'être supprimée. Vous ne dessinerez pas les frames d'impression cout.

```
#include <iostream>
using namespace std;

struct point3D {
    double x;
    double y;
    double z;
};

struct point3D milieuSegment(struct point3D pointA, struct point3D pointB) {
    struct point3D milieu;
    milieu.x = (ptA.x + ptB.x)/2.0;
    milieu.y = (ptA.y + ptB.y)/2.0;
    milieu.z = (ptA.z + ptB.z)/2.0;
    return milieu;
}

int main() {
    struct point3D pointM = {0.0, 2.0, 0.0};
    struct point3D pointN = {1.0, 0.0, 2.0};
    struct point3D mil = milieuSegment(pointM, pointN);
    cout << "Le milieu du segment [MN] a pour coordonnees (x=" << mil.x << ",
y=" << mil.y << ", z=" << mil.z << endl;
    return 0;
}
```

Exercice 4 : Appel à une fonction qui prend un tableau en entrée

- a. Donnez la trace mémoire et la trace écran du programme suivant. Utilisez le modèle de pile vu en cours. Vous supposerez que la valeur de retour du programme principal est stockée à l'adresse 3 976 628 994 et qu'un int occupe 4 octets. Vous dessinerez une « photo » de la pile à chaque fois qu'une frame est sur le point d'être supprimée. Vous ne dessinerez pas les frames d'impression cout.

```
void afficheTableau (const int tab[], int taille) {
    for (int i = 0; i < taille; i++)
        cout << tab[i] << " ";
}

int main() {
    int notes[5] = {12,10,14,5,8};
    afficheTableau(notes,5);
    return 0;
}
```

- b. Expliquez à quoi sert le mot-clé const dans l'entête de la procédure afficheTableau.

Exercice 5 : Procédure récursive sur un tableau

Considérons le programme C++ suivant :

```
#include <iostream>
using namespace std;

void mystere(double t[], int a, int b) {
    double aux;
    if (a < b) {
        aux = t[a];
        t[a] = t[b];
        t[b] = aux;
        mystere(t, a+1, b-1);
    }
    else {
        /* DESSIN : état de la mémoire quand on passe dans ce else */
    }
}

int main() {
    double monTab[4] = {9.0, 10.0, 11.0, 12.0};
    int i;

    mystere(monTab, 0, 3);

    cout << "Tableau apres traitement :\n";
    for (i = 0; i < 4; i++)
        cout << "monTab[" << i << "] = " << monTab[i] << endl;

    return 0;
}
```

- Que fait la procédure mystere ? Autrement dit, si vous deviez lui donner un nom plus explicite, lequel choisiriez-vous ?
- Dessinez l'état de la pile au moment indiqué en commentaire. Vous utiliserez le modèle théorique de pile vu en cours et en TD. Vous supposerez que la valeur de retour du main est stockée à l'adresse 3 987 546 988.

Exercice 6 : Tableaux de structures et fonction récursive

Soit le programme C++ suivant :

```
#include <iostream>
#include <math.h>
using namespace std;

struct sPoint {
    double x;
    double y;
};
typedef struct sPoint Point;

double dist (Point p1, Point p2) {
```

```

    return sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y));
}

double longueurchemin (const Point * chemin, int nb) {
/* Préconditions : chemin est un tableau contenant au moins nb points,
    avec nb supérieur ou égal à 2
    Résultat : retourne la longueur du chemin de points
*/
    double res;
    double d1, d2;

    if (nb==2) res = dist(chemin[1],chemin[0]);
    else {
        d1 = dist(chemin[nb-1],chemin[nb-2]);
        d2 = longueurchemin(chemin,nb-1);
        res = d1 + d2;
    }
    return res;
}

int main() {
    double perimetre;
    Point cheminTriangle[4];
    cheminTriangle[0].x = 0.0 ;
    cheminTriangle[0].y = 0.0 ;
    cheminTriangle[1].x = 3.0 ;
    cheminTriangle[1].y = 0.0 ;
    cheminTriangle[2].x = 3.0 ;
    cheminTriangle[2].y = 1.0 ;
    cheminTriangle[3].x = cheminTriangle[0].x ;
    cheminTriangle[3].y = cheminTriangle[0].y ;

    perimetre = longueurchemin(cheminTriangle, 4);
    cout << "Le perimetre du triangle vaut " << perimetre << endl;

    return 0;
}

```

- Combien d'octets une variable de type Point occupe-t-elle en mémoire (sur une machine où un int occupe 4 octets et un double 8) ?
- Combien d'octets occupe le tableau cheminTriangle ?
- La fonction longueurchemin est-elle récursive ? Même question pour la fonction dist.
- A quoi sert le mot-clé const dans l'entête de la fonction longueurchemin ?
- Dessinez l'évolution de la pile lors de l'exécution de ce programme, en utilisant le modèle théorique de pile vu en cours et en TD. Vous supposerez que la valeur de retour du main est stockée à l'adresse 3 987 546 998.
- Combien de fois la fonction longueurchemin est-elle appelée ? Même question pour la fonction dist.
- Que pensez-vous de l'implémentation récursive de longueurchemin, en termes de ressources mémoire ?
- Quelle valeur obtenez-vous pour la variable perimetre ? Indication : $\sqrt{10} \approx 3.162$

TD2 : Vie et mort des variables en mémoire (2/2)

Exercice 1 : Pointeurs et tableaux en C++, arithmétique des pointeurs

```
int monTab[] = {-25, -6, 8, 15, 38, 50, 72, 81, 98};  
  
int * p = monTab;
```

Quelles valeurs ou adresses fournissent les expressions suivantes ?

- a. `*p+2`
- b. `*(p+2)`
- c. `p+2`
- d. `&p`
- e. `&monTab[4] - 3`
- f. `monTab+3`
- g. `&monTab[7] - p`
- h. `*(p+8) - monTab[7]`

Exercice 2 : Pointeurs et allocation dynamique de mémoire

```
#include <iostream> /* entrées-sorties avec cin et cout */  
using namespace std;  
  
int main() {  
    int e = 10;  
    double r = 3.14;  
    int * p1;  
    double * p2;  
    int i;  
  
    p1 = new int [5];  
    if (p1 == NULL) { cout << "Allocation ratee \n"; exit(1); }  
  
    p2 = new double;  
    if (p2 == NULL) { cout << "Allocation ratee \n"; exit(1); }  
  
    for (i=0; i < 5; i++) { p1[i] = e-i; }  
    *p2 = r;  
    /* Faire la trace mémoire (1) */  
    (*p1)*=5;  
    *p2=p1[3]*2.0;  
    /* Faire la trace mémoire (2) */  
    e = 25;  
    r = -8.3;  
    /* Faire la trace mémoire (3) */  
    delete [] p1;  
    delete p2;
```



```

/* Faire la trace mémoire (4) */
return 0;
}

```

- Expliquez ce que signifie l'instruction `p1 = new int [5];`
- Proposez une autre façon d'écrire l'instruction `(*p1) *= 5;`
- Faites les quatre traces mémoire de ce programme, en supposant que la valeur de retour du main est stockée à l'adresse 0xD4986C36 (soit 3 566 758 966 en base 10) et qu'il n'y a pas de problème d'allocation mémoire.

Exercice 3 : Appel à une fonction qui retourne un tableau

Dessinez l'état de la pile et du tas aux endroits indiqués en commentaires. Vous utiliserez le modèle théorique de pile vu en cours et en TD. Vous supposerez que la valeur de retour du main est stockée à l'adresse 3 987 546 988.

```

/* Precondition: tab1 et tab2 sont de même taille */
/* Postcondition: de la mémoire est allouée dans le tas, charge à l'utilisateur de la
libérer quand il n'en a plus besoin */
float * sommeTab(const float * tab1, const float * tab2, const int taille) {
    int i;
    float * resultat = new float [taille];
    for (i=0; i < taille; i++) resultat[i] = tab1[i] + tab2[i];
    return resultat;
    /* Faire la trace mémoire */
}

int main() {
    float notes1[] = {8.5, 12.6, 14.5, 10.0, 9.1};
    float notes2[] = {12.2, 5.8, 17.3, 11.7, 7.6};
    float * somme = NULL;

    somme = sommeTab(notes1, notes2, 5);
    /* Faire la trace mémoire */
    delete [] somme;
    return 0;
}

```

Exercice 4 : Appel à une procédure qui modifie un pointeur

Soit le programme C++ suivant :

```

double deux_fois (const double * i) {return 2*(*i);}

double * allouer_en_initialisant (double val) {
    double *p = new double;
    *p = val;
    return p;
}

void liberer_en_mettant_a_zero (double ** pa) { delete (*pa); *pa = NULL; }

int main() {
    double e = 3.1;
    double k;
    double * p1;
    double tab[4];
}

```

```

double * p2;
double **pp2;

k = deux_fois(&e);
p1 = allouer_en_initialisant(-8.0);

for (p2=tab; p2 < tab+4; p2++) { *p2 = (*p1) + k; }
p2 = &e;
liberer_en_mettant_a_zero(&p1);

pp2 = &p2;
return 0;
}

```

- Pour chaque apparition du caractère '*', indiquez s'il s'agit d'une déclaration de pointeur, de l'opérateur de déréférencement d'un pointeur, ou bien de l'opérateur de multiplication.
- Faites la trace mémoire de ce programme, en supposant que la valeur de retour du main est située à l'adresse 3 678 556 960.
- Pourquoi passe-t-on un « pointeur sur pointeur » (double **, c'est-à-dire l'adresse d'une adresse de double) à la procédure libérer_en_mettant_a_zero ?

Exercice 5 : Appel à des fonctions manipulant des objets

Soit le programme C++ suivant :

```

class Point3D {
public:
    float x,y,z;
}

float calculerNorme (const Point3D & p) {
    float norme = sqrt(p.x*p.x + p.y*p.y + p.z*p.z);
    return norme;
}

Point3D normaliser (const Point3D p) {
    float norme;
    Point3D resultat;
    norme = calculerNorme(&p);
    resultat.x = p.x / norme; resultat.y = p.y / norme; resultat.z = p.z / norme;
    return resultat;
}

int main() {
    Point3D point;
    point.x = 3.0; point.y = 4.0; point.z = 0.0;
    Point3D pointNormalise;
    pointNormalise = normaliser(point);
    return 0;
}

```

Dessinez l'état de la pile au cours de l'exécution de ce programme. Vous utiliserez le modèle théorique de pile vu en cours et en TD. Vous supposerez que la valeur de retour du main est stockée à l'adresse 3 987 546 988.

Exercice 6 : Allocation dynamique d'objets

Soit le programme C++ suivant :

```
class Point3D {
public:
    float x,y,z;
}

Point3D * calculerMilieu (const Point3D & p1, const Point3D * p2) {
    Point3D * milieu;
    milieu = new Point3D ;
    milieu->x = (p1.x+p2->x)/2.0;
    milieu->y = (p1.y+p2->y)/2.0;
    milieu->z = (p1.z+p2->z)/2.0;
    return milieu;
}

int main() {
    Point3D point1, point2;
    point1.x = 5.0; point1.y = 6.0; point1.z = -1.0;
    point2.x = 3.0; point2.y = -2.0; point2.z = 3.0;
    Point3D * pointMilieu = NULL;
    pointMilieu = calculerMilieu(point1,&point2);
    delete pointMilieu;
    return 0;
}
```

Dessinez l'état de la pile au cours de l'exécution de ce programme. Vous utiliserez le modèle théorique de pile vu en cours et en TD. Vous supposerez que la valeur de retour du main est stockée à l'adresse 3 987 546 988.

TD3 : Classes et objets

Exercice 1 : Conception et spécificateur

On souhaite créer un type de donnée pour représenter une personne. Cette personne est identifiée par son nom, son prénom et son numéro de sécurité sociale (numéro unique). On veut pouvoir saisir et afficher ces informations.

- Ecrivez, en notation algorithmique, la classe Personne.
- Traduisez cette classe en C++.
- Donner le programme principal, en C++, permettant de saisir puis d'afficher les informations d'une personne.
- Que doit-on faire pour interdire aux utilisateurs de la classe de manipuler les données membres directement?
- Afin de permettre leur manipulation, écrivez des procédures/fonctions dont le rôle est de lire et modifier les données membres. Quels sont les avantages et inconvénients de cette conception ?
- Ecrivez un programme principal qui modifie le nom d'une personne après saisie.

Exercice 2 : Constructeur et destructeur en mémoire

Soit le programme C++ suivant :

```
#include <iostream>
using namespace std;

class Point2D {
public:
    float x,y;
    Point2D () {x = 0.0; y = 0.0; cout << "Point2D nul créé\n";}
    Point2D (float _x, float _y) {
        x = _x; y = _y;
        cout << "Point2D créé\n";
    }
    Point2D (const Point2D & p) {
        x = p.x; y = p.y;
        cout << "Point2D copié\n";
    }
    ~Point2D () {cout << "Point2D détruit\n";}
}

int main() {
    Point2D pt1;
    Point2D pt2 (1.0,2.5);
    Point2D * ppt3 = new Point2D (pt2);
    delete ppt3;
    return 0;
}
```

- Dessiner l'état de la mémoire après chaque construction d'objet en supposant que la valeur de retour du main est stockée à l'adresse 3 987 546 988 et qu'il n'y a pas de problème d'allocation mémoire.
- Donner la trace écran de l'exécution de ce programme.

- c. Au lieu de définir un constructeur sans paramètre et un autre avec comme paramètres les données membres, vous pouvez utiliser des valeurs par défaut. Ecrivez un constructeur avec paramètres par défaut pouvant remplacer les deux autres.

Exercice 3 : Appel à des fonctions membres

Soit le programme C++ suivant :

```
#include <iostream>
using namespace std;

class Point2D {
public:
    float x,y;
    Point2D (float _x, float _y) {x = _x; y = _y;}
    ~Point2D () {}

    float distanceOrigine() {return sqrt(x*x+y*y);}
    float distancePoint (const Point2D & p) {
        return sqrt((p.x-x)*(p.x-x)+(p.y-y)*(p.y-y));
    }
}

int main() {
    Point2D pt1 (2.6,7.5);
    Point2D pt2 (1.4,3.8);
    float distpt10;
    float distpt2pt1;
    distpt10 = pt1.distanceOrigine();
    distpt2pt1 = pt2.distancePoint(pt1);
    return 0;
}
```

- a. Dessiner l'état de la mémoire avant les suppressions de frame (sauf constructeurs) en supposant que la valeur de retour du main est stockée à l'adresse 3 987 546 988.
- b. La fonction distanceOrigine n'est finalement qu'un cas particulier de distancePoint. Réécrivez le code de distanceOrigine afin d'utiliser le code de distancePoint.

Exercice 4 : Surcharge de fonctions membres

On souhaite développer une classe gérant les entrées/sorties standards (clavier et écran). Cette classe doit pouvoir afficher des entiers, des réels, des chaînes de caractères et des points 2D (selon le format : (x , y)) en utilisant un nom de procédure unique. De même, la lecture se fera par une fonction de nom unique.

- a. Ecrire la classe ES gérant les entrées/sorties de ces types.
- b. Donner un programme principal lisant et affichant une variable de chaque type.

Exercice 5 : Surcharge d'opérateurs

On souhaite finalement pouvoir faire des opérations mathématiques sur des points 2D (cf. exercices précédents). Une solution serait de définir des fonctions membres telles que **void** additionnerPoints(**const** Point2D &

p). Une autre solution consiste à définir les opérateurs élémentaires pour un point 2D de telle façon à ce que l'on puisse exécuter des instructions telles que `pt3=pt1+pt2;`.

- a. Ecrire la classe Point2D avec les fonctionnalités nécessaires pour additionner, soustraire et tester l'égalité de deux points.
- b. Ajouter les fonctionnalités d'affectation et d'incrément préfixé (ajout de 1 à chaque coordonnée lorsque l'on fait `++pt`).
- c. Ajouter les fonctionnalités de symétrie centrale par rapport à l'origine définie par l'opérateur unaire « - », et le test si le point est situé à l'origine avec l'opérateur « ! ».
- d. Ajouter les fonctionnalités d'affichage et de lecture des coordonnées du point suivant le format `(x , y)`.

TD4 : Algorithmes, invariants de boucle et complexité

Exercice 1 : Algorithme de chiffrement d'une chaîne de caractères

Considérons l'algorithme de chiffrement de chaînes de caractères dit « de Vigenère ». Il consiste à « additionner » les caractères du texte à chiffrer avec ceux d'une clé de chiffrement. Par exemple, le chiffrement de la chaîne « Cherchez au pied de l'arbre » avec la clé « indice » peut s'illustrer ainsi :

- on place la clé en regard du texte à chiffrer, en répétant la clé autant de fois que nécessaire pour couvrir le texte, et en ignorant les caractères qui ne sont pas des lettres (ils seront laissés inchangés par l'algorithme de chiffrement) :

C	h	e	r	c	h	e	z		a	u		p	i	e	d		d	e		l	'	a	r	b	r	e
i	n	d	i	c	e	i	n		d	i		c	e	i	n		d	i		c	'	e	i	n	d	i

- on remplace chaque lettre de la clé par sa position dans l'alphabet (0 pour 'a', 1 pour 'b'...),

C	h	e	r	c	h	e	z		a	u		p	i	e	d		d	e		l	'	a	r	b	r	e
8	13	3	8	2	4	8	13		3	8		2	4	8	13		3	8		2	'	4	8	13	3	8

- on remplace chaque lettre du texte à chiffrer par la lettre située d positions plus loin dans l'alphabet, où d est le nombre indiqué par la clé (si on dépasse z, on reboucle sur a, b etc.) :

C	h	e	r	c	h	e	z		a	u		p	i	e	d		d	e		l	'	a	r	b	r	e
K	u	h	z	e	l	m	m		d	c		r	m	m	q		g	m		n	'	e	z	o	u	m

Notez qu'une lettre identique dans le texte à chiffrer ne produit pas forcément le même code (ex. le premier et deuxième e de Cherchez donnent respectivement un h et un m). Et notez qu'un même code ne donne n'est pas forcément issu d'une même lettre (ex. les deux m successifs du troisième mot sont issus d'un i et d'un e). Cela rend le processus de décryptage très difficile sans la clé.

- Si x est le code ASCII d'une lettre à chiffrer et y le code ASCII de la lettre de la clé située en regard, quelle formule permet de calculer le code ASCII résultant ? On supposera pour cette question que les deux lettres sont des minuscules. Indice : que vaut $y - 'a'$?
- Ecrire en langage algorithmique la procédure de chiffrement, dont voici l'entête :

Procédure chiffrer (texte : chaîne de caractères, cle : chaîne de caractères, result : chaîne de caractères)
Préconditions : texte contient un ou plusieurs caractères suivis d'un caractère de terminaison '\0'. cle contient une ou plusieurs lettres minuscules suivie de '\0'. result est une chaîne déjà allouée en mémoire, assez grande pour contenir le texte crypté (incluant la place pour le caractère de terminaison).
Postcondition : result contient la version cryptée de texte. Seules les lettres (majuscules ou minuscules) non accentuées sont cryptées, les autres caractères sont recopiés tels quels.
Paramètres en mode donnée : texte, cle
Paramètre en mode donnée-résultat : result

- Supposons que l'on veuille chiffrer, avec une clé de longueur k , une chaîne constituée exclusivement de L minuscules, avec $L > k$. Comptez le nombre d'opérations de chaque type (affectations d'entiers, comparaison de caractères, etc.) pour évaluer la complexité en temps de cet algorithme de chiffrement. Qui de L ou de k influence le plus le temps d'exécution ? Ce temps sera-t-il plus long ou plus court avec une plus longue clé ?
- Quel serait le prototype de la procédure en C++ ?

Exercice 2 : Tri mystère

Voici le début de la trace d'un algorithme de tri :

10	56	-2	52	-8	41	13
-8	56	-2	52	10	41	13
-8	-2	56	52	10	41	13
-8	-2	10	52	56	41	13

- De quel algorithme s'agit-il ? Pourquoi ?
- Complétez le tableau ci-dessus en indiquant les étapes suivantes, jusqu'à ce que le tableau soit complètement trié.
- Si le tableau est de taille n , quelle est la complexité de cet algorithme en fonction de n ? Justifiez (brièvement) votre réponse.

Exercice 3 : Analyse de la complexité d'un algorithme

On considère le pseudo-code suivant, comportant deux « tant que » imbriqués. On cherche à mesurer la complexité de cette imbrication en fonction de n . Pour cela, on utilise la variable « compteur », qui est incrémentée à chaque passage dans le « tant que » interne.

Variables :

n : entier

compteur : entier

i, j : entiers

Début

Afficher(« Quelle est la valeur de n ? »)

Saisir(n)

compteur $\leftarrow 0$

$i \leftarrow 1$

Tant que ($i < n$) Faire

$j \leftarrow i + 1$

 Tant que ($j \leq n$) Faire

 compteur \leftarrow compteur + 1

$j \leftarrow j + 1$

 Fin tantque

$i \leftarrow i * 2$

Fin tantque

Afficher(compteur)

Fin

- Quelle est la valeur finale du compteur dans le cas où $n = 16$?
- Considérons le cas particulier où n est une puissance de 2 : on suppose que $n = 2^p$ avec p connu. Quelle est la valeur finale du compteur en fonction de p ? Justifiez votre réponse.
- Réexprimez le résultat précédent en fonction de n .

Exercice 4 : Tri à bulles

Le tri à bulles est un algorithme de tri qui s'appuie sur des permutations répétées d'éléments contigus qui ne sont pas dans le bon ordre.

Procédure tri_bulles(tab : tableau [1..n] de réels)

Précondition : tab est un tableau contenant n réels

Postcondition : les éléments de tab sont triés dans l'ordre croissant

Paramètre en mode donnée : aucun

Paramètre en mode donnée-résultat : tab

Variables locales : i, j : entiers, e : réel

Début

```
1  i ← 1
2  Tant que (i ≤ n-1) Faire
3      j ← n
4      Tant que (j ≥ i+1) Faire
5          Si tab[j] < tab[j-1] Alors
6              {on permute les deux éléments}
7              e ← tab[j-1]
8              tab[j-1] ← tab[j]
9              tab[j] ← e
10         Fin Si
11         j ← j-1
12     Fin Tant que
13     i ← i + 1
14 Fin Tant que
```

Fin tri_bulles

- Soit le tableau suivant : {53.8, 26.1, 2.5, 13.6, 8.8, 4.0}. Donnez les premiers états intermédiaires par lesquels passe ce tableau lorsqu'on lui applique la procédure tri_bulles.
- Complétez la phrase suivante de sorte à ce qu'elle corresponde à l'invariant de boucle du Tant que interne (boucle sur j, lignes 4 à 12 de l'algorithme).

« Lorsqu'on vient de décrémenter j (ligne 11), le _____ du sous-tableau tab[_____] se trouve en position _____. De plus, si j>1, les éléments du sous-tableau tab[_____] occupent les mêmes positions qu'avant le démarrage de la boucle sur j. »

- Déduisez-en la propriété que présente le tableau lorsqu'on a terminé cette boucle interne, c'est-à-dire lorsqu'on arrive sur la ligne 13.
- En utilisant la propriété précédente, on peut montrer que l'invariant de boucle du Tant que externe (boucle sur i) est le suivant : « Juste avant d'incrémenter i (ligne 13) : tab est trié entre les indices 1 et i, et tous les éléments restants sont supérieurs ou égaux à tab[i] ». Donnez la première étape de cette démonstration (initialisation). Les étapes de conservation et de terminaison ne sont pas demandées.
- Calculez le nombre total d'affectations de réels réalisées par la procédure tri_bulles lors du tri complet d'un tableau de n réels, dans le cas le plus défavorable.

Exercice 5 : Tri par insertion

Le tri par insertion est l'algorithme utilisé par la plupart des joueurs lorsqu'ils trient leur « main » de cartes à jouer. Le principe consiste à prendre le premier élément du sous-tableau non trié et à l'insérer à sa place dans la partie triée du tableau.

- Dérouler le tri par insertion du tableau {5.1, 2.4, 4.9, 6.8, 1.1, 3.0}.
- Ecrire en langage algorithmique le corps de la procédure de tri par insertion, par ordre croissant, d'un tableau de réels :

Procédure tri_par_insertion (tab : tableau [1..n] de réels)

Précondition : tab[1], tab[2], ... tab[n] initialisés

Postcondition : $\text{tab}[1] \leq \text{tab}[2] \leq \dots \leq \text{tab}[n]$

Paramètres en mode donnée : aucun

Paramètre en mode donnée-résultat : tab

- Donner l'invariant de boucle correspondant à cet algorithme, en démontrant qu'il vérifie bien les 3 propriétés d'un invariant de boucle : initialisation, conservation, terminaison.
- Evaluer le nombre de comparaisons de réels et le nombre d'affectations de réels pour un tableau de taille n , dans le cas le plus défavorable (tableau trié dans l'ordre décroissant). Cet algorithme est-il meilleur que le tri par sélection (ou tri du minimum), vu en cours magistral ?

Exercice 6 : Stabilité d'un algorithme de tri

Considérons la liste de noms-prénoms suivante : Doe, Phil

Doe, Jane

Doe, Fred

Jones, Bill

Jones, Jane

Smith, Mary

Smith, Fred

Smith, Jane

Elle est pour l'instant triée dans l'ordre alphabétique des noms de famille. Mais on veut la trier dans l'ordre alphabétique des prénoms, à l'aide du tri ci-dessous.

Procédure tri (tab : tableau [1...n] de Personnes)

Précondition : tab[1], tab[2], ... tab[n] initialisés

Postcondition : $\text{tab}[1] \leq \text{tab}[2] \leq \dots \leq \text{tab}[n]$

Paramètres en mode donnée : aucun

Paramètre en mode donnée-résultat : tab

Variables locales :

i, j : entiers

elt_a_placer : Personne

Début

Pour i allant de 2 à n par pas de 1 Faire

elt_a_placer \leftarrow tab[i]

j \leftarrow i - 1

```

    Tant que (j > 0) et (elt_a_placer.prenom est strictement
                    avant tab[j].prenom dans l'alphabet) Faire
        tab[j+1] ← tab[j]
        j ← j - 1
    Fin TantQue
    tab[j+1] ← elt_a_placer
    Fin Pour
Fin tri_par_insertion

```

- De quel algorithme de tri s'agit-il ?
- Quelle liste obtient-on au final avec ce tri ?
- Un algorithme de tri est dit « stable » s'il préserve toujours l'ordre initial des ex-aequo. Dans notre exemple, l'algorithme est stable si les personnes avec le même prénom (qui vont être groupées ensemble lors du tri par prénom) restent dans l'ordre alphabétique des noms de famille. L'algorithme de tri par insertion est-il stable ? (La preuve n'est pas demandée).
- Combien de comparaisons de prénoms ferait-on avec cet algorithme s'il était appelé sur un tableau de taille n qui serait déjà dans le bon ordre (c'est-à-dire, ici, déjà trié dans l'ordre alphabétique des prénoms) ?

Exercice 7 : Calcul d'un polynôme en un point

Soit un polynôme P tel que $P(x) = \sum_{k=0}^n a_k x^k$.

On veut écrire la fonction qui retourne la valeur de $P(x)$ pour une valeur de x passée en paramètre, les coefficients du polynôme étant également passés en paramètre dans un tableau. Pour cela, une méthode efficace est la méthode de Horner, qui utilise la réécriture suivante de $P(x)$:

$$P(x) = (((...((a_n x + a_{n-1})x + a_{n-2})x + \dots)x + a_1)x + a_0$$

La méthode consiste donc à multiplier le coefficient de plus haut degré par x et à lui ajouter le coefficient suivant. On multiplie alors le nombre obtenu par x et on lui ajoute le troisième coefficient, etc., jusqu'à avoir ajouté le coefficient constant.

Exemple : calcul de $4x^3 - 7x^2 + 3x - 5 = ((4x - 7)x + 3)x - 5$
pour $x = 2$:

- première étape : $4 \cdot 2 - 7 = 1$
- deuxième étape : $1 \cdot 2 + 3 = 5$
- troisième étape : $5 \cdot 2 - 5 = 5$

Voici le code C++ de la fonction qui calcule $P(x)$ selon cette méthode :

```

/* Précondition : coef contient les (n+1) coefficients du polynôme, ainsi coef[0]
    contient a0, coef[1] contient a1, etc.
   Résultat : retourne P(x), la valeur du polynôme au point x */
double valeur_polynome (double x, double coef [], int n) {
1   double y = 0.0;
2   int k = n;
3   while (k >= 0) {
4       y = y * x + coef[k];

```

```

5     k = k - 1;
6 }
7 return y;
8 }

```

- Combien de fois passe-t-on par les lignes 4 et 5 si le polynôme est de degré n ?
- Complétez le tableau suivant en fonction de n , le degré du polynôme. Vous prendrez bien sûr en compte le nombre de fois que l'on passe sur chaque ligne lors de l'exécution complète de la fonction.

Ligne	Nombre d'affectations	Nombre d'additions ou de soustractions	Nombre de multiplications	Nombre de comparaisons
1				
2				
3				
4				
5				
Total				

- Déduisez-en le temps d'exécution $T(n)$ de la fonction en microsecondes, en supposant qu'une affectation prend t_{aff} microsecondes, une addition ou une soustraction t_{add} microsecondes, une multiplication t_{mult} microsecondes, et une comparaison t_{comp} microsecondes.
- De quelle fonction mathématique de n ce temps d'exécution $T(n)$ est-il « grand O » ?
- Complétez l'invariant de boucle de l'algorithme de Horner : « Au début de chaque itération de la boucle while (juste avant d'exécuter la ligne 4), on sait que (complétez les pointillés) : $y = \sum_{i=0}^{n-(k+1)} a_{\dots} x^i$ ».
- A l'aide la propriété de terminaison de cet invariant, montrez que l'algorithme de Horner permet bien d'obtenir la valeur du polynôme au point x .
- Ecrivez en langage C++ la version « naïve » de la fonction valeur_polynome, qui calcule $P(x)$ selon la formule $P(x) = \sum_{k=0}^n a_k x^k$, c'est-à-dire en calculant pour chaque terme la puissance de x via autant de multiplications que nécessaire (on n'utilisera donc pas la fonction mathématique `pow()`).
- Combien de multiplications fait-on dans cette version naïve, si le polynôme est de degré n ?

Exercice 8 : Tri par fusion (interne)

Ecrire une version itérative (c'est-à-dire non récursive) de l'algorithme du tri par fusion d'un tableau de réels. L'analyse de complexité n'est pas demandée ici car elle sera vue au TD suivant.

TD5 : Algorithmes sur les fichiers

Exercice 1 : Fusion de deux monotonies sur fichiers

On dispose de deux fichiers contenant chacun une séquence triée de réels. En d'autres termes, chacun des deux fichiers contient une monotonie et une seule – il s'agit donc d'un problème plus simple que celui du tri fusion sur fichiers (cf. exercice 2). On veut écrire une procédure qui fusionne ces deux monotonies et écrit la monotonie résultante dans un troisième fichier. Cette procédure prend comme *seuls* paramètres les noms des deux fichiers d'entrée (**nomfic1** et **nomfic2**) et le nom du fichier de sortie (**nomficSortie**).

Préconditions :

- nomfic1 et nomfic2 sont des fichiers texte qui contiennent chacun une séquence triée (monotonie) de nombres au format IEEE 754 double précision. Si l'un de ces fichiers ne peut pas être ouvert (par exemple parce qu'il n'existe pas), alors le programme se termine avec un code d'échec.
- Les deux séquences peuvent être de longueur différente, mais elles sont au moins de longueur 1 : autrement dit, chaque fichier contient au moins un élément.

Postconditions :

- Un nouveau fichier nommé nomficSortie est créé (s'il existait déjà, son contenu est écrasé). Ce fichier contient une monotonie correspondant à la fusion des deux monotonies contenues dans nomfic1 et nomfic2. Si ce fichier ne peut pas être créé (par exemple à cause d'un problème de droits insuffisants dans le répertoire courant), le programme se termine avec un code d'échec.
- Les contenus des fichiers nomfic1 et nomfic2 sont inchangés.

Donnez le code C++ de cette procédure. Vous utiliserez les variables locales suivantes (à vous d'en préciser le type) : **e1**, **e2**, **fic1**, **fic2**, **ficSortie**, **succeslect1**, **succeslect2**. Vous pouvez en ajouter d'autres si nécessaire.

Exercice 2 : Tri fusion sur fichier

On considère le fichier non trié contenant la séquence d'éléments suivante :

65	5	89	56	7	15	28	2	98	33
----	---	----	----	---	----	----	---	----	----

- Donnez le contenu des 3 fichiers (appelés A, B et X dans les diapositives de cours) intervenant dans le tri fusion du fichier ci-dessus, après chaque étape d'éclatement et chaque étape de fusion.
- Combien de comparaisons d'éléments effectue-t-on au pire lorsqu'on fusionne une monotonie de longueur L_a avec une autre monotonie de longueur L_b ?
- Combien de comparaisons d'éléments effectue-t-on au pire lorsqu'on trie par fusion un fichier de n éléments, dans le cas particulier où n est une puissance de 2 ($n = 2^n$) ?
- Ecrire en langage algorithmique la procédure de tri par fusion d'un fichier, en supposant que vous disposez déjà des procédures « éclatement » et « fusion » (voir les entêtes ci-dessous). La procédure de tri doit appeler les procédures « éclatement » et « fusion » jusqu'à ce que le fichier soit complètement trié.

Procédure éclatement (nomFicX : chaîne de caractères, lg : entier, nomFicA : chaîne de caractères, nomFicB : chaîne de caractères)

Précondition : le fichier appelé nomFicX contient des monotonies de longueur lg, sauf peut-être la dernière qui peut être plus courte

Postconditions : Les monotonies de nomFicX sont réparties (1 sur 2) dans des fichiers appelés nomFicA et nomFicB : la première monotonie est copiée dans le fichier A, la seconde dans le fichier B, la troisième dans

le A, etc. Si ces fichiers existaient déjà, leur contenu est écrasé. Le fichier A peut recueillir une monotonie de plus que le fichier B, et cette dernière monotonie peut être de longueur inférieure à lg. Si au contraire ficA et ficB recueillent le même nombre de monotopies, la dernière monotonie écrite dans ficB peut être de longueur inférieure à lg.

Paramètres en mode donnée : nomFicX, nomFicA, nomFicB, lg

Remarque : Les chaînes de caractères contenant les noms des fichiers ne vont pas être affectées par la procédure, d'où le passage en mode donnée, mais les fichiers désignés par nomFicA et nomFicB vont être affectés, comme cela est précisé dans les post-conditions.

Procédure fusion (nomFicA : chaîne de caractères, nomFicB : chaîne de caractères, lg : entier, nomFicX : chaîne de caractères, nbMonoDansX : entier)

Préconditions : les fichiers appelés nomFicA et nomFicB contiennent des monotopies de longueur lg. FicA peut contenir une monotonie de plus (éventuellement incomplète) que FicB. Si au contraire FicA et FicB contiennent le même nombre de monotopies, alors la dernière monotonie de FicB peut être incomplète.

Postconditions : le fichier appelé nomFicX contient nbMonoAprès monotopies de longueur $2 \times lg$, la dernière pouvant être plus courte. Ces monotopies résultent de la fusion 2 à 2 des monotopies de FicA et de FicB. Si FicX existait déjà, son ancien contenu est écrasé.

Paramètres en mode donnée : nomFicX, nomFicA, nomFicB, lg

Paramètre en mode donnée-résultat : nbMonoDansX

- e. Donnez le code de la procédure d'éclatement en langage C++.

Exercice 3 : Compression RLE

Soit un fichier binaire de n octets pouvant contenir des séquences répétitives d'un même octet. Par exemple, en décodant chaque octet comme un « unsigned char » :

{12, 6, 6, 6, 6, 6, 1, 3, 8, 8, 10, 2, 53, 53, 53, 53, 6, 6, 6, 13}

Une première technique de compression consiste à écrire pour chaque octet le nombre de répétitions supplémentaires. On pourrait donc remplacer {53, 53, 53, 53, 53} par {53, 4}, ce qui signifie : l'octet 53, suivi d'encore 4 fois l'octet 53. Le problème de cette première idée est que la séquence {53, 53, 53, 53, 53} va bien être compressée, mais qu'au contraire, toutes les séquences non répétitives vont être dilatées : par exemple, {10} serait réécrit en {10, 0}.

Une meilleure idée consiste à ne transformer une séquence que si un octet est immédiatement répété au moins une fois. Ainsi, {53, 53, 53, 53, 53} serait réécrite en {53, 53, 3}, {10} serait inchangée et {8, 8} serait réécrite en {8, 8, 0}. La séquence entière donnée en exemple serait donc réécrite de la façon suivante : {12, 6, 6, 3, 1, 3, 8, 8, 0, 10, 2, 53, 53, 3, 6, 6, 1, 13}. Lors de la décompression, c'est le fait d'avoir deux octets identiques à la suite qui indique que l'octet suivant est un nombre d'occurrences supplémentaires et non un octet du fichier de départ. **C'est à cette seconde idée que vous allez vous intéresser dans tout cet exercice.** Il s'agit de la compression dite « RLE » (Run Length Encoding). Notez que bien que ce second algorithme soit la plupart du temps meilleur que le premier, il n'est pas parfait pour autant : il existe tout de même des cas pour lesquels il dilate au lieu de compresser. Il favorise les successions de plus de trois caractères identiques, il défavorise les successions de deux et laisse inchangé les simples occurrences.

- a. Supposons que le fichier d'entrée contient 10 octets. Donnez un exemple de contenu de fichier qui donnerait la plus petite taille possible pour le fichier de sortie, puis un exemple de contenu qui donnerait la plus grande taille possible en sortie.

Considérons la procédure C++ suivante, qui compresse un fichier binaire selon l'algorithme RLE.

```
void compresser (const string & nomFichierEntree, const string & nomFichierSortie) {
1  unsigned char octet_courant, octet_prec, nb_occ = 1, nb_occ_suppl = 0;
2  ifstream entree(nomFichierEntree.c_str()); if (!entree.is_open()) exit(EXIT_FAILURE);
3  ofstream sortie(nomFichierSortie.c_str()); if (!sortie.is_open()) exit(EXIT_FAILURE);
4  if (entree.eof()) {
5      /* Fichier d'entrée vide, on laisse le fichier de sortie vide */
6      entree.close();
7      sortie.close();
8      return;
9  }
10 entree.read((char*)(&octet_prec), sizeof(octet_prec)); // lecture binaire de octet_prec
11 sortie.write((const char*)(&octet_prec), sizeof(octet_prec)); // et son écriture binaire
12
13 while (!entree.eof()) {
14     entree.read((char*)(&octet_courant), sizeof(octet_courant));
15     if (octet_courant != octet_prec) { // nouveau caractère lu
16         if (nb_occ > 1) { // le précédent caractère a plusieurs occurrences
17             nb_occ_suppl = nb_occ - 2;
18             sortie.write((const char*)(&octet_prec), sizeof(octet_prec)); //doublement du carac.
19             sortie.write((const char*)(&nb_occ_suppl), sizeof(nb_occ_suppl)); //écriture nb_occ-2
20             nb_occ = 1; // retour au caractère courant dont c'est la première occurrence
21         }
22         sortie.write((const char*)(&octet_courant), sizeof(octet_courant)); // écriture courant
23         octet_prec = octet_courant;
24     }
25     else { // relecture du même caractère
26         nb_occ++;
27         if (nb_occ == 255) { // voir question c
28             nb_occ_suppl = nb_occ - 2;
29             sortie.write((const char*)(&octet_prec), sizeof(octet_prec));
30             sortie.write((const char*)(&nb_occ_suppl), sizeof(nb_occ_suppl));
31             nb_occ = 1;
32             octet_prec += 5; /* modification arbitraire de octet_prec
33                               pour être dans le bon cas à l'itération suivante */
34         }
35     }
36 }
37
38 if (nb_occ > 1) { // si le fichier se termine par une séquence répétitive
39     nb_occ_suppl = nb_occ - 2;
40     sortie.write((const char*)(&octet_prec), sizeof(octet_prec));
41     sortie.write((const char*)(&nb_occ_suppl), sizeof(nb_occ_suppl));
42 }
43 entree.close();
44 sortie.close();
45 }
```

b. Considérons un fichier d'entrée contenant n octets avec n pair et non nul, et dont le contenu correspondrait au cas le pire en termes de taux de compression (voir la question a). Combien fait-on d'opérations de chaque type suivant lors de l'exécution de la procédure compresser ?

- Nombre d'affectations d'unsigned char :
- Nombre d'additions d'unsigned char :
- Nombre de soustractions d'unsigned char :
- Nombre de comparaisons d'unsigned char :
- Nombre d'appels à read :

- Nombre d'appels à write :
 - Nombre d'appels à eof :
- c. A quoi servent les lignes 27 à 34 ?
- d. Donner le code C++ de la procédure de décompression, dont l'entête est la suivante :

```
void decompresser (const string & nomFichierEntree, const string & nomFichierSortie);  
/*  
Préconditions : nomFichierEntree est le nom d'un fichier binaire compressé selon  
l'algorithme RLE.  
Postconditions : un nouveau fichier nommé comme spécifié dans nomFichierSortie est  
créé, son contenu correspond à la décompression du fichier d'entrée.  
*/
```


TD6 : Exercices de synthèse

Exercice 1 : Débogage

Le fichier source suivant contient plusieurs « bugs » que vous allez devoir localiser.

```
#include <iostream>
using namespace std;

double moyenne (double tab[]) {
    int i;
    double moy = 0.0;
    for (i = 0; i < nb; i++) {moy += tab[i];}
    return moy/nb;
}

int main() {
    string prenom;
    double * lesNombres;
    int i, nb;

    cout << "Entrez votre prenom :\n";
    cin >> prenom;
    cout << "Bonjour, " << prenom << " !\n";

    do {
        cout << "Entrez le nombre d'elements du tableau \n";
        cout << "(ce nombre doit etre un entier superieur ou egal a 1) \n";
        cin >> nb;
    } while (nb < 1);

    lesNombres = new double [nb];
    cout << "Saisissez les " << nb << " valeurs :\n";
    for (i = 0; i < nb; i++) { cin >> lesNombres[i]; }
    cout << "La moyenne des valeurs est " << moyenne(lesNombres) << endl;

    cout << "Au revoir " << prenom;
    return 0;
}
```

- Ce code ne compile pas. Expliquez pourquoi et indiquez la ou les modifications qu'il faut apporter au code pour qu'il compile.
- Ce code comporte une fuite de mémoire. Expliquez pourquoi et indiquez la ou les modifications qu'il faut apporter au code pour corriger cette fuite.

Exercice 2 : Trace mémoire avec procédure récursive

Considérons le programme C++ suivant :

```
#include <iostream>
#include <string>
using namespace std;
```

```

void mystere (unsigned int i, unsigned int nbr, string & t) {
    unsigned int reste;
    if (nbr != 0) {
        reste = nbr % 16;
        if (reste <= 9) {t[i] = reste + '0';}
        else {t[i] = reste - 10 + 'a';}
        mystere(i-1, nbr / 16, t);
    }
    else {
        /* Dessin 1 = état de la mémoire quand on rentre dans ce else */
    }
}

int main() {
    string monTab ("00000000");
    char * c = NULL;
    unsigned int k = 8;
    unsigned int monNombre = 455;

    mystere(7, monNombre, monTab);
    cout << monTab << endl;

    c = new char [9];
    for (k = 0; k < 9; k++) {*(c + k) = monTab[k];}
    /* Dessin 2 = état de la mémoire à ce moment */

    delete [] c;
    return 0;
}

```

- Dessinez l'état de la pile et du tas aux deux moments indiqués en commentaire (on vous demande pour chaque dessin une « photo » de la pile entière et de tout ce qu'il y a dans le tas). Vous utiliserez le modèle théorique de pile vu en cours et en TD. Vous supposerez que la valeur de retour du main est stockée à l'adresse 3 987 546 988.
- Indiquez ce qu'affiche le cout du main lorsqu'on exécute ce programme précis.
- Que fait la procédure mystere ? Autrement dit, si vous deviez lui donner un nom plus explicite, lequel choisiriez-vous ?
- Si les unsigned int sont codés sur 32 bits, risque-t-on un « buffer overflow » lorsqu'on remplit monTab avec la procédure mystère ?

Exercice 3 : Trace mémoire, tri et invariant de boucle

```

#include <iostream>
using namespace std;

class NbComplexe {
public:
    double a;
    double b;
    bool operator > (const NbComplexe & n) {
        /* this est supérieur à n si son module est plus grand */
        double m, mn;
    }
}

```

```

    m = a*a + b*b;
    mn = n.a*n.a + n.b*n.b;
    /* Qc : Dessinez l'état de la mémoire à chaque fois que l'on passe ici */
    if (m > mn) return true;
    else return false;
}
NbComplexe& operator = (const NbComplexe & n) {a = n.a; b = n.b; return *this;}
};

void tri (NbComplexe * t, int n) {
    int i, j;
    NbComplexe z;
    bool rescomp;

    for (j = 1; j < n; j++) {
        /* Qb : Indiquez quel est l'invariant de boucle à ce moment */
        z = t[j];
        i = j - 1;
        rescomp = t[i] > z;
        while ((i >= 0) && rescomp) {
            t[i+1] = t[i];
            i--;
            if (i >= 0) rescomp = t[i] > z;
        }
        t[i+1] = z;
    }
}

int main() {
    int s = 3;
    NbComplexe * mesNb = new NbComplexe [s];
    mesNb[0].a = 2.0; mesNb[0].b = 4.0;
    mesNb[1].a = 0.0; mesNb[1].b = 1.0;
    mesNb[2].a = 1.0; mesNb[2].b = 2.0;
    (*(mesNb + 1)).b = 0.0;

    tri(mesNb, s);

    delete [] mesNb;
    return 0;
}

```

- De quel algorithme de tri s'agit-il ?
- Quel est son invariant de boucle ? Vous utiliserez les notations du code ci-dessus (j, n, etc.) et vous considérerez que le tableau est indexé de 0 à n-1.
- Dessinez l'état de la pile et du tas aux moments indiqués en commentaire (on vous demande pour chaque dessin une « photo » de la pile entière et de tout ce qu'il y a dans le tas). Vous utiliserez le modèle théorique de pile vu en cours et en TD. Vous supposerez que la valeur de retour du main est stockée à l'adresse 3 987 546 988.

TD7 : Tableaux dynamiques

Exercice 1 : Rappels

- Quelle est la spécificité d'un tableau dynamique ?
- Rappeler les données membres de la classe utilisée pour l'implémentation d'un tableau dynamique.
- Donner le code C++ du constructeur de la classe utilisée pour l'implémentation d'un tableau dynamique. Ce constructeur prendra en paramètre la taille initiale du tableau à allouer.

Exercice 2 : Dynamique des données en mémoire

Expliquez pourquoi la précondition de l'initialisation d'un tableau dynamique est nécessaire. Pour illustrer votre propos, voici un exemple de programme principal en langage algorithmique qui ne respecte pas cette précondition : faites un schéma de ce qu'il se passe en mémoire lorsqu'on l'exécute. Pour faire ce schéma, vous pouvez supposer que les éléments sont simplement des réels.

Importer

```
module TableauDynamique  
module ElementTD
```

Début

```
monTab : TableauDynamique  
monElem : ElementTD  
i : entier  
initialiser(monTab)  
initialiser(monElem, 17.3)  
Pour i allant de 1 à 5 par pas de 1  
    ajouterElement(monTab, monElem)  
Fin Pour  
initialiser(monTab)  
{travail demandé = photo de la mémoire à ce moment du programme}  
testament(monTab)
```

Fin

Exercice 3 : Crible d'Eratosthène et décomposition en facteurs premiers

- On se propose de calculer tous les nombres premiers plus petits qu'un entier n donné. La méthode consiste à calculer pas à pas ces nombres en utilisant la règle suivante : si un entier k n'est divisible par aucun nombre premier plus petit que k alors il est lui-même premier. Quelles sont les structures de données qu'on peut utiliser pour résoudre ce problème ? Quelle est la plus efficace ?
- Ecrire en C++ la procédure **eratosthene** qui calcule les nombres premiers plus petits que n passé en paramètre.
- Ecrire en C++ la procédure **decompose**, qui détermine les nombres premiers entrant dans la décomposition d'un entier n .

- d. On veut écrire un fichier Makefile pour mettre au point un module appelé **NombresPremiers** qui fournit les procédures eratosthene et decompose, et un exécutable qui permet de tester ce module. Commencez par lister les dépendances de chaque fichier .o et de l'exécutable, puis écrivez le Makefile.

Exercice 4 : Complexité de l'extension et notion de coût amorti

En cours, nous avons fait le choix de doubler la capacité du tableau à chaque fois qu'une extension s'avère nécessaire. Aurait-on eu la même performance si l'on avait choisi d'augmenter la capacité du tableau de 10 emplacements au lieu de la doubler ?

Exercice 5 : Recherche dichotomique dans un tableau initialement trié

- a. Ecrire en C++ la fonction membre qui renvoie l'indice, de l'élément `e` passé en paramètre, dans un tableau trié. Cette recherche se fera de façon dichotomique.
- b. Quel est le coût d'une telle recherche ?

Exercice 6 : Suppression d'un élément dans un tableau dynamique

On veut maintenant libérer de la place lorsqu'on en a de trop. En effet, lorsqu'on supprime un ou plusieurs éléments dans un tableau, la taille utilisée peut devenir très petite par rapport à la taille réservée. Dans ce cas on va libérer « un peu » de place pour n'utiliser que ce qu'il faut.

- a. Réfléchissez à la méthode à employer.
- b. Ecrivez la procédure membre `supprimerElement` qui tient compte de la réduction de la capacité d'un tableau dynamique.

Exercice 7 : Procédure de recopie

- a. Que se passe-t-il lorsqu'on fait `a = b` quand `a` et `b` sont deux tableaux dynamiques ?
- b. Ecrire en C++ l'opérateur d'affectation d'un tableau dynamique initialisé dans un autre tableau déjà initialisé également.

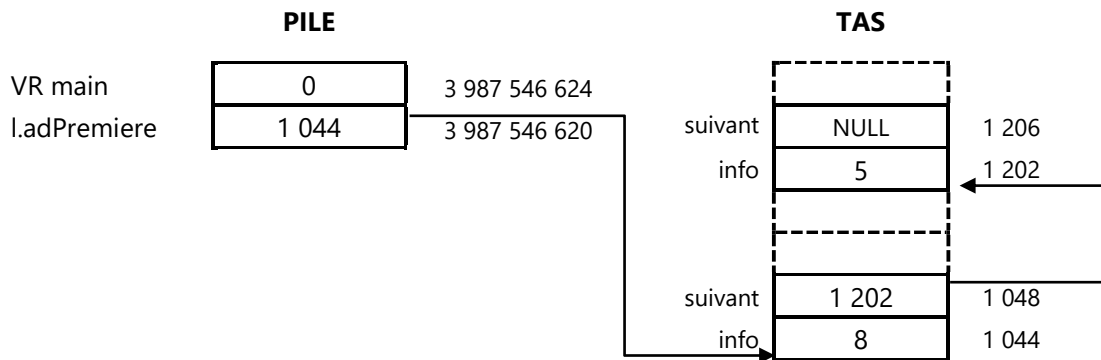
TD8 : Listes chaînées

On supposera dans tout ce TD que les listes sont simplement chaînées.

Exercice 1 : Dynamique des données en mémoire

Considérons le programme principal et l'état de la mémoire correspondant :

```
int main () {  
    Liste l;  
    l.ajouterEn.....(8);  
    l.ajouterEn.....(5); /* Etat de la mémoire dessiné à ce moment */  
    ..... /* Instruction à rajouter*/  
    ..... /* Instruction à rajouter*/  
    return 0;  
}
```



- Utilisez la représentation graphique présentée en cours pour décrire la liste créée par ce programme.
- Complétez les pointillés des appels `l.ajouterEn.....(8)` et `l.ajouterEn.....(5)`.
- Rajoutez dans le programme principal les deux instructions qui permettent :
 - d'afficher l'information contenue dans la première cellule.
 - d'afficher l'information contenue dans la deuxième cellule.

Exercice 2 : Affichage d'une liste

Ecrivez la procédure membre `afficher` qui affiche dans l'ordre le contenu d'une liste (les éléments sont séparés par un caractère d'espace).

Exercice 3 : Création d'une liste à partir d'un tableau

Écrivez un constructeur de la classe `Liste` qui, à partir d'un tableau dynamique d'éléments, crée la liste contenant les mêmes éléments dans le même ordre. Donnez un exemple d'appel à ce constructeur. On supposera que `ElementTD` et `ElementL` sont des types compatibles.

Exercice 4 : Inversion d'une liste

Ecrire en C++ une procédure qui inverse l'ordre des éléments d'une liste chaînée passée en paramètre, sans faire de delete ni de new.

Exercice 5 : Fusion de deux listes

Ecrire en C++ une procédure qui fusionne deux listes triées passées en paramètres. Le résultat de la fusion sera stocké dans une troisième liste, triée elle aussi. Les deux listes de départ sont inchangées.

Exercice 6 : Tri d'une liste chaînée par insertion

Ecrire en C++ une procédure qui trie une liste chaînée passée en paramètre en utilisant l'algorithme de tri par insertion.

Exercice 7 : Suppression des occurrences d'un élément

- a. Ecrire une procédure membre qui supprime **la première occurrence** d'un élément e d'une liste.
- b. Ecrire une procédure membre qui supprime **toutes les occurrences** d'un élément e dans une liste.

TD9 : Piles et Files

Exercice 1 : Fonctionnement de base d'une pile et d'une file

Dans cet exercice, on suppose que les éléments sont des caractères. Donner l'état de la pile et de la file ainsi que les valeurs des autres variables après l'exécution des instructions suivantes :

Début	Début
p : Pile	f : File
s : ElementP	t : ElementF
v : booléen	v : booléen
p.initialiserPile()	f.initialiserFile()
p.empiler('a')	f.enfiler('a')
p.empiler('k')	f.enfiler('k')
p.empiler('z')	f.enfiler('z')
p.dépiler()	f.dépiler()
p.empilerPile('m')	f.enfiler('m')
s <- p.consulterSommet()	t <- f.premierDeLaFile()
v <- p.estVide()	v <- f.estVide()
p.testamentPile()	f.testamentFile()
Fin	Fin

Exercice 2 : Inversion d'une File en utilisant une Pile

Le but de cet exercice est d'écrire (en langage algorithmique) une procédure qui inverse une file d'entiers qui lui est passée en paramètre. On demande de ne pas utiliser de tableau ou de liste de travail pour effectuer l'inversion, mais d'utiliser plutôt une pile. Il existe en effet une méthode très simple pour inverser une file en utilisant une pile.

Exercice 3 : Validité du parenthésage d'une expression

Un problème fréquent pour les compilateurs et les traitements de textes est de déterminer si les parenthèses d'une chaîne de caractères sont équilibrées et proprement incluses les unes dans les autres. On désire donc écrire une fonction qui teste la validité du parenthésage d'une expression :

- on considère que les expressions suivantes sont valides : "()", "[([bonjour+]essai)7plus-];"
- alors que les suivantes ne le sont pas : "(", ")", "4(essai]"

Notre but est donc d'évaluer la validité d'une expression en ne considérant que ses parenthèses et ses crochets. On suppose que l'expression à tester est dans une chaîne de caractères, dont on peut ignorer tous les caractères autres que '(', '[', ']' et ')'. Écrire en langage algorithmique la fonction valide(ch : chaîne de caractères) : booléen qui renvoie vrai si l'expression passée en paramètre est valide, faux sinon.

Exercice 4 : Notation polonaise

La notation polonaise, ou encore notation post-fixée, consiste à faire précéder les opérandes des opérateurs. Par exemple, au lieu d'écrire $42 + 13$, en notation polonaise on écrit $42\ 13\ +$. Un des avantages de cette notation est qu'elle rend inutile l'usage des parenthèses : pour $3 \times (42 + 13) - 5$, on note $3\ 42\ 13\ +\ \times\ 5\ -$ et il n'y a aucune ambiguïté. Dans cet exercice on va définir une fonction `polonaise` qui prendra en paramètres une expression post-fixée sous la forme d'un tableau de chaînes de caractères `({"3","42","13","+","*","5","-"})` pour l'exemple précédent) et sa taille, et renverra le résultat (numérique) de l'évaluation de cette expression.

L'algorithme est le suivant. On lit les éléments du tableau un par un et on les empile sur une pile initialement vide. A chaque fois qu'on rencontre un opérateur, plutôt que de l'empiler, on l'applique aux deux derniers éléments de la pile et le résultat remplace ces deux derniers éléments.

- Créer une fonction `estOperateur` qui prend une chaîne de caractères en paramètre et retourne vrai si cette chaîne représente un opérateur valide (ici seulement `"+"`, `"-"`, `"/"` ou `"*"`), et faux sinon.
- Créer une fonction `calcul` qui prend en paramètre des chaînes de caractères représentant un opérateur `op` et deux opérandes `a` et `b`, et qui retourne la valeur `a op b`. Vous pourrez utiliser la fonction de conversion suivante : `double atof (const char * c);`
- En utilisant les deux fonctions précédentes, écrire la fonction `polonaise`. Vous pourrez convertir un réel en une chaîne de caractères en utilisant les instructions suivantes :

```
ostringstream ss; ss << monReel; string ch (ss.str());
```

TD10 : Arbres binaires

Exercice 1 : Hauteur d'un arbre binaire

Ecrire en langage C++ la fonction qui détermine la hauteur d'un arbre binaire passé en paramètre.

Exercice 2 : Egalité de deux arbres binaires

Ecrire en langage algorithmique une fonction qui teste l'égalité de deux arbres binaires.

Exercice 3 : Dessiner des arbres binaires de recherche

Dessiner des ABR de hauteur 2, 3, 4, 5 et 6 pour le même ensemble de clés {1, 4, 5, 10, 16, 17, 21}.

Exercice 4 : Parcours préfixé itératif d'un arbre binaire

Ecrire en langage C++ la procédure **itérative** du parcours préfixé dans un arbre binaire.

Exercice 5 : Recherche itérative dans un arbre binaire quelconque

Ecrire en langage C++ une fonction **itérative** qui cherche un élément dans un arbre binaire quelconque. Comparez le coût de cette recherche avec celui de la recherche dans un ABR.

Exercice 6 : Tester si un arbre binaire est un arbre binaire de recherche

Ecrire en langage C++ une fonction qui renvoie vrai si un arbre binaire passé en paramètre est un arbre binaire de recherche et faux sinon.

Exercice 7 : Suppression dans un ABR

Ecrire en langage algorithmique la procédure de suppression d'un élément dans un arbre binaire de recherche.

Travaux pratiques

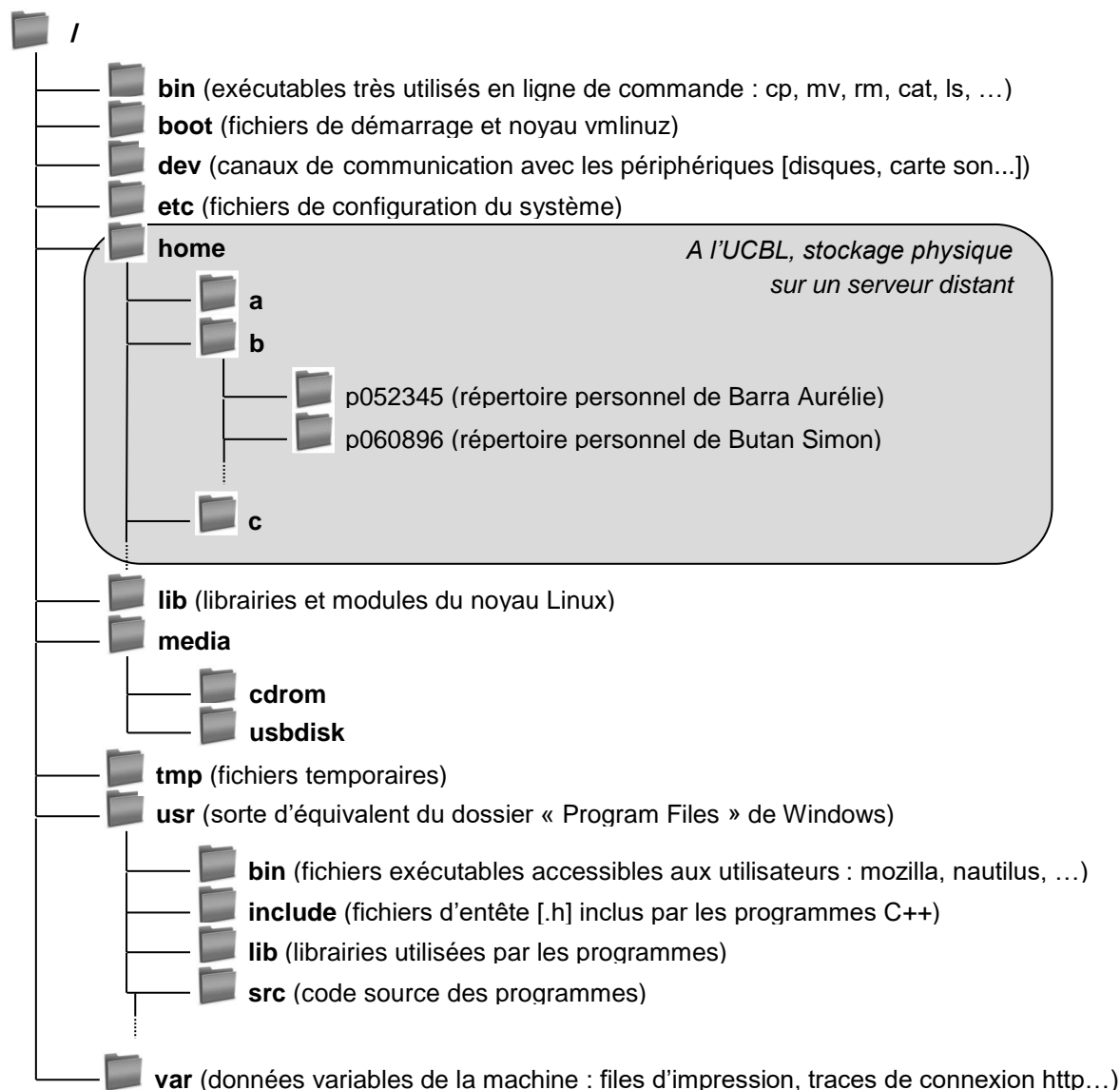
Notes :

[illegible]

TP1 : De LIFAP1 à LIFAP3...

Exercice 1 : De Windows à Linux

Redémarrez l'ordinateur sous Linux (Ubuntu). Connectez-vous avec les mêmes login / mot de passe que sous Windows. Pour utiliser au mieux son compte Linux, il est nécessaire de connaître quelques notions basiques sur le système de fichiers Linux. La racine du système de fichier (l'équivalent du « C : \ » d'un Windows non partitionné) est le répertoire « / ». Voici un schéma des principaux répertoires que contient ce dossier racine.



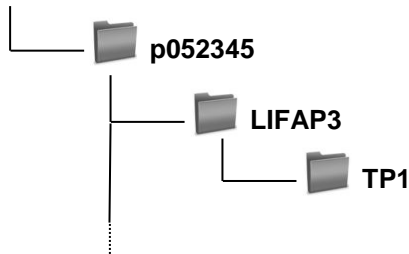
A l'université, les répertoires des utilisateurs ne sont pas stockés physiquement sur les machines des salles de TP, mais sur un serveur auquel les machines de TP accèdent par le réseau (même principe que pour votre lecteur W: sous Windows). Mais cela est transparent pour l'utilisateur, qui accède toujours à son répertoire personnel en allant dans `/home/[1ère lettre du nom de famille]/[n° d'étudiant]`. Cela permet d'avoir accès à ses données même si l'on change de poste de travail (ou d'OS).

Vous pouvez consulter le contenu de votre compte utilisateur de deux façons :

- avec le gestionnaire de fichiers Nautilus (menu Applications / Dossier personnel)

- en ligne de commande :
 - ouvrez un terminal (menu / Emulateur de terminal)
 - vérifiez que vous êtes dans votre répertoire personnel en tapant `pwd` (« print working directory »)
 - demandez le listing du contenu du répertoire en tapant `ls -a`

Dans cette UE, nous allons privilégier l'utilisation de la ligne de commande. Voir l'annexe A pour les commandes Linux de base et des précisions sur la notion de chemin sous Linux. Vous allez créer l'arborescence suivante dans votre répertoire personnel, en n'utilisant que la ligne de commande.



Pour cela, allez dans le terminal, puis :

- vérifiez que vous êtes dans votre répertoire personnel en tapant `pwd`. Si vous n'y êtes pas, retournez-y en tapant `cd` (cela signifie « change directory », et si l'on ne précise pas de répertoire de destination, on va par défaut dans le répertoire personnel)
- créez le répertoire LIFAP3 en tapant la commande suivante (respectez bien l'espace après mkdir, mais n'en mettez pas dans le nom du répertoire) : `mkdir LIFAP3`
- vérifiez que ce nouveau répertoire apparaît dans le répertoire courant, en tapant `ls`
- allez dans le répertoire créé en tapant `cd LIFAP3`
- créez le répertoire TP1 en tapant `mkdir TP1` et déplacez-vous dans ce répertoire

Vérifiez que vous retrouvez bien les dossiers créés en explorant votre dossier personnel en mode graphique.

Exercice 2 : De CodeBlocks aux outils minimaux (éditeur de texte, gcc, terminal)

En LIFAP1, vous avez programmé en C/C++ à l'aide de CodeBlocks, qui est un « environnement de développement intégré » regroupant un éditeur de texte, un compilateur, des outils automatiques de fabrication, et un débogueur. Il en existe d'autres, par exemple Dev-C++. Cependant, programmer dans un environnement plus minimaliste permet d'apprendre à distinguer les éléments essentiels d'un programme et d'une chaîne de compilation. Pour programmer en C++, il faut au minimum : un éditeur de texte, un compilateur en ligne de commande, et un terminal dans lequel on tape les commandes de compilation et d'exécution. C'est ce que nous allons faire dans cette UE. Cela vous permettra, par la suite, de mieux comprendre ce que fait un environnement de développement lorsque vous cliquez sur un bouton « Compiler », par exemple. Cela vous permettra aussi de mieux comprendre les mystérieux fichiers « Makefile » utilisés par ces environnements, car vous allez en faire vous-même (à partir du TP5).

Nous allons utiliser l'éditeur de texte « gedit » (mais vous pouvez aussi utiliser un autre éditeur comme « Kate »). Pour cela, allez dans le terminal, puis :

- Vérifiez que vous êtes dans le répertoire TP1 en tapant `pwd`. Si vous n'y êtes pas, retournez-y en tapant `cd ~/LIFAP3/TP1` (~ désigne votre répertoire personnel).
- Lancez gedit en tapant `gedit hello.cpp &`. Comme le fichier hello.cpp n'existe pas encore, gedit le crée pour vous (fichier vide que vous allez pouvoir remplir). Ajouter & en fin de commande vous permet de reprendre la main dans le terminal. Cela va vous permettre en autres de taper les commandes de compilation et d'exécution tout en gardant la fenêtre avec le code ouverte.
- Tapez le code suivant dans gedit (fichier hello.cpp) :

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello world !\n";
    return 0;
}
```

- Sauvegardez vos modifications sans fermer gedit. Retourner dans le terminal (vous aurez peut-être à taper entrée si gedit a envoyé des messages sur la console), et taper `ls` pour vérifier que le nouveau fichier nommé `hello.cpp` est apparu dans le répertoire TP1.
- Nous allons maintenant compiler ce code à l'aide du compilateur gcc. Il s'agit du principal compilateur C++ libre. Pour compiler, tapez :

```
g++ -g -Wall -o hello.out hello.cpp
```

- Vérifiez, en tapant `ls`, qu'un nouveau fichier nommé `hello.out` est apparu dans le répertoire TP1.
- Tapez `man g++` pour comprendre ce que signifient les différents éléments de la commande de compilation, et complétez le tableau suivant (aide : une fois la documentation affichée, tapez `/mot` pour rechercher un mot, `n` pour passer à l'occurrence suivante, `q` pour sortir).

g++	
-g	
-Wall	
-o hello.out	
hello.cpp	

- Exécutez le programme en tapant `./hello.out` dans le terminal.
- Toujours dans le terminal, appuyez plusieurs fois sur la flèche vers le haut. Que se passe-t-il ? A quoi cela peut-il servir ?

Exercice 3 : Structures et fonctions globales

Dans cet exercice vous allez définir une structure pour des vecteurs en trois dimensions et des fonctions manipulant ces vecteurs. Dans gedit, ouvrez un nouveau fichier que vous appellerez `Vecteur3D.cpp`. Vous avez à présent deux fichiers ouverts dans gedit : `hello.cpp` et `Vecteur3D.cpp`. Lorsque l'on travaille sur plusieurs fichiers en même temps, il est nettement préférable d'avoir une seule fenêtre gedit et d'utiliser les onglets, plutôt que d'ouvrir plusieurs fenêtres gedit.

- Dans le fichier `Vecteur3D.cpp`, définir une structure `Vecteur3D` qui contient trois champs de réels (x, y et z).
- Puis ajouter les procédures et fonctions suivantes
 - une fonction `Vecteur3DGetNorme` qui retourne la norme d'un vecteur passé en paramètre
 - une procédure `Vecteur3DNormaliser` qui normalise le vecteur passé en paramètre
 - une fonction `Vecteur3DEstNormalise` qui indique si le vecteur passé en paramètre est normalisé
 - une fonction `Vecteur3DAdd` qui retourne le vecteur somme de deux vecteurs passés en paramètre

- une procédure `Vecteur3DAfficher` qui affiche à l'écran le vecteur passé en paramètre sous le format : (x,y,z)

c. Recopier le programme principal suivant en fin de fichier

```
int main () {
    Vecteur3D vecteur1 = {5,2,1};
    Vecteur3D vecteur2 = {0,3,2};

    cout << "vecteur1 non normalise: ";
    Vecteur3DAfficher(vecteur1);
    cout << endl;

    cout << "vecteur2 non normalise: ";
    Vecteur3DAfficher(vecteur2);
    cout << endl;

    cout << "somme: ";
    Vecteur3DAdd(vecteur1,vecteur2);
    cout << endl;

    Vecteur3DNormaliser(vecteur1);
    Vecteur3DNormaliser(vecteur2);

    cout << "vecteur1 normalise: ";
    Vecteur3DAfficher(vecteur1);
    cout << endl;

    cout << "vecteur2 normalise: ";
    Vecteur3DAfficher(vecteur2);
    cout << endl;

    cout << "somme: ";
    Vecteur3D somme = Vecteur3DAdd(vecteur1,vecteur2);
    Vecteur3DAfficher(somme);
    if (Vecteur3DEstNormalise(somme)) cout << " est normalise" << endl;
    else cout << " n'est pas normalise" << endl;
    return 0;
}
```

d. Compiler et exécuter le programme (nommer l'exécutable `Vecteur3D.out`). Vérifier que vos procédures et fonctions fonctionnent correctement en vérifiant que la trace écran correspond à ce que vous attendez.

Exercice 4 : Tableau de structures et paramètre

Toujours dans le fichier `Vecteur3D.cpp`, ajouter les procédures et fonctions suivantes :

- Une procédure `Vecteur3DRemplirTabVecteurs` qui remplit un tableau déjà alloué en mémoire passé en paramètre avec des vecteurs dont les coordonnées sont des valeurs aléatoires comprises entre -10 et 10 (avec une décimale). La taille du tableau est aussi passée en paramètre. Consulter l'annexe B pour apprendre comment générer aléatoirement des valeurs.
- Une procédure `Vecteur3DAfficherTabVecteurs` qui affiche à l'écran l'ensemble des vecteurs contenus dans un tableau de vecteurs passé en paramètre sous le format : `vec1 ; vec2 ; ... ; vecn`. La taille du tableau est aussi passée en paramètre.
- Une fonction `Vecteur3DMaxTabVecteurs` qui retourne l'indice du vecteur de plus grande norme dans un tableau de vecteurs passé en paramètre. La taille du tableau est aussi passée en paramètre.

▶

Ecrire le programme principal qui crée un tableau de 10 vecteurs aux valeurs réelles aléatoires, l'affiche, puis affiche le vecteur de plus grande norme. Compiler et exécuter votre programme, et vérifier que vos procédures et fonctions fonctionnent correctement en vérifiant que la trace écran correspond à ce que vous attendez.

TP2 : Vie et mort des variables en mémoire

Exercice 1 : Modèle de pile pour l'adressage

Reprenez le fichier `Vecteur3D.cpp` que vous avez créé au TP précédent et ajoutez-y des « cout » pour visualiser les adresses mémoires des paramètres et des variables locales des différents sous-programmes.

Aide : pour afficher l'adresse d'une variable, vous pouvez la présenter en tant que long int :

```
cout << "Adresse de monFloat : " << (long int) &monFloat;
```

Vous pouvez aussi demander la taille en octets occupée par une variable ou un type avec l'opérateur `sizeof` :

```
cout << "Taille occupée par monTab : " << sizeof(monTab);
```

Comparez l'évolution théorique de la pile avec ce qui se passe en réalité :

- Dans quel ordre sont empilés les éléments d'un tableau ? La case 0 a-t-elle l'adresse la plus haute ou la plus basse ?
- Dans quel ordre sont empilés les paramètres d'une fonction ou d'une procédure ?
- Dans quel ordre sont empilées les données membres d'un objet ?
- Les variables locales d'un même sous-programme sont-elles, comme en TD, dans l'ordre dans lequel elles étaient déclarées dans le code ?
- Lorsque l'un des paramètres d'un sous-programme est un tableau, combien d'octets ce paramètre occupe-t-il dans la frame du sous-programme ? Le tableau est-il recopié dans cette frame ?
- Quel écart observez-vous entre l'adresse la plus haute et l'adresse la plus basse, parmi les adresses affichées ? Cet écart correspond-il à l'écart théorique (celui de votre trace « papier ») ? Demandez à votre encadrant d'où vient cette différence.

Exercice 2 : Modèle de pile pour les appels récursifs et visualisation des frames avec gdb

- Tapez le programme de l'exercice 1 du TD1 (appel à une fonction récursive) dans un nouveau fichier. Compilez-le et exécutez-le pour vérifier que tout fonctionne bien.
- Puis recompilez-le en ajoutant à gcc l'option `-g` (cette option ajoute à l'exécutable des informations de débogage), et lancez le debugger gdb avec la commande `gdb nomDeVotreExécutable`. Placez un point d'arrêt sur la ligne 10 (correspondant au `return 1;`) en tapant `break 10`. gdb vous informe que ce breakpoint est le numéro 1, on pourrait en mettre d'autres si on le souhaitait. Lancez ensuite l'exécution du programme en tapant `run` ou juste `r`. L'exécution va s'arrêter lorsqu'on va entrer dans le if, juste avant d'exécuter le `return`.
- Demandez à gdb la liste des « frames » actives en mémoire à ce moment en tapant `backtrace`. Combien de frames voyez-vous pour comb ? Que se passe-t-il donc quand une fonction s'appelle elle-même : réutilise-t-on la même frame ou en empile-t-on une nouvelle à chaque appel ?
- gdb identifie les frames par des numéros (frame #0, frame #1, etc...). Sélectionnez la frame 0 pour l'examiner, en tapant `select-frame 0`. Demandez ensuite à gdb quels sont les paramètres de cette frame et quelles sont les valeurs courantes de ses variables locales en tapant `info args` puis `info locals`. Pourquoi les valeurs de `tmp1` et `tmp2` sont-elles étranges ?
- Recommencez pour une autre frame. Les valeurs des paramètres sont-elles les mêmes pour des frames différentes d'une même fonction ?

- f. Quittez gdb en tapant `quit`.

Exercice 3 : Allocation dynamique de mémoire dans le main

Ecrivez dans un nouveau fichier un programme qui demande à l'utilisateur de taper la taille qu'il souhaite pour son tableau, alloue un tableau de réels de la taille demandée, et demande à l'utilisateur les valeurs des réels à stocker dans le tableau. Pour la lecture des saisies clavier, vous pourrez utiliser `cin`. Le programme affichera ensuite le tableau et se terminera proprement, c'est-à-dire en libérant la mémoire allouée dynamiquement.

Exercice 4 : Trois entêtes pour une même fonction

Reprenez la fonction `comb` de l'exercice 2 (c'est-à-dire de l'exercice 1 du TD1). Vous allez faire deux implémentations supplémentaires de cette fonction de combinaison qui vont différer par leurs entêtes, c'est-à-dire par la façon dont les entrées et les sorties sont gérées.

- Dans le fichier écrit à l'exercice 2, indiquez les pré- et post-conditions en commentaires de la fonction déjà réalisée.
- Ajouter une procédure de même nom (`comb`) qui prend en paramètre `n` et `p` mais aussi le résultat du calcul de la combinaison en mode donnée-résultat. Indiquez en commentaires les pré- et post-conditions de cette procédure. Ajouter au main un appel à cette procédure avec les mêmes valeurs de `n` et `p` que précédemment et vérifier que vous obtenez le même résultat (l'afficher).
- Ajouter une autre procédure de même nom (`comb`) qui prend en paramètre `n` et `p` et un pointeur sur un entier alloué sur le tas dans lequel vous mettrez le résultat du calcul. Indiquez en commentaires les pré- et post-conditions de cette procédure. Ajouter au main un appel à cette procédure avec les mêmes valeurs de `n` et `p` que précédemment et vérifier que vous obtenez le même résultat (l'afficher).

TP3 : Classes et objets

Dans ce TP, nous allons écrire un programme qui va trier des tableaux de nombres complexes.

Exercice 1 : Définition du type nombre complexe

- Définissez le type `NbComplexe` à l'aide d'une classe. Rappel : un nombre complexe est défini par sa partie réelle (que vous pourrez appeler `re`) et sa partie imaginaire (que vous pourrez appeler `im`).
- Ajoutez deux fonctions membres à cette classe : une pour saisir le nombre complexe au clavier (initialisation des données membres) et une pour afficher le nombre complexe à l'écran sous le format : `re +im i` si la partie imaginaire est positive ou nulle, et `re -im i` si elle est négative.
- Créez un main qui crée un nombre complexe sur la pile, l'affiche, puis le saisit et le réaffiche. Qu'obenez-vous ?
- Quelle instruction permet d'afficher la taille, en octets, d'un nombre complexe ?

Exercice 2 : Constructeur, destructeur et allocation dynamique

- Ajoutez à la classe `NbComplexe` trois constructeurs et un destructeur. Le premier constructeur sera sans paramètre, le deuxième aura deux paramètres pour les deux parties du nombre complexe, et le troisième est un constructeur par copie.
- Ajoutez une procédure membre `multiplier` qui multiplie le nombre complexe par un autre nombre complexe passé en paramètre. L'instance courante contient le résultat de la multiplication, le nombre complexe avec lequel on multiplie n'est pas modifié.

$$\text{Rappel : } (re_1 + im_1 i) \times (re_2 + im_2 i) = (re_1 \times re_2 - im_1 \times im_2) + (im_1 \times re_2 + re_1 \times im_2) i$$

- Complétez le main en ajoutant la création d'un nouveau nombre complexe comme étant une copie du nombre saisi à l'exercice précédent (affichez le pour vérifier que votre constructeur fonctionne correctement). Ensuite créez un nombre complexe différent **sur le tas** et affichez le. Finalement, multipliez les deux nombres et affichez le résultat.

Exercice 3 : Comparaison de deux nombres complexes

Ajoutez dans la classe `NbComplexe` les deux fonctions membres suivantes :

- une fonction `module` qui retourne le module du nombre complexe. Rappel : $|re + im i| = \sqrt{re^2 + im^2}$
- une fonction `estPlusPetit` qui indique si le nombre complexe est plus petit qu'un autre nombre complexe passé en paramètre. La comparaison se fera sur les valeurs des modules des deux nombres.

Testez ces deux fonctions sur quelques nombres complexes.

Exercice 4 : Création d'un tableau de nombres complexes aléatoires

- Dans le main, allouez de la mémoire pour un tableau de nombres complexes dont la taille sera saisie par l'utilisateur.

- b. Remplissez le tableau avec des nombres complexes dont les parties réelles et imaginaires sont tirées aléatoirement dans l'intervalle $[-10,10]$ (avec 1 décimale de précision). L'annexe B décrit comment faire un tirage aléatoire d'une valeur entière dans un intervalle.
- c. Affichez le tableau de nombres complexes ainsi rempli et pour chaque élément du tableau affichez également le module du nombre complexe. N'oubliez pas de libérer la mémoire allouée dynamiquement quand vous en n'avez plus besoin.

Exercice 5 : Tri par sélection du tableau de nombres complexes

Définissez une procédure globale (pas une procédure membre) `trierParSelection` qui prend en paramètres un tableau de nombres complexes et sa taille, et qui le trie du nombre le plus petit au plus grand (en termes de module), en utilisant l'algorithme de tri par sélection. Testez votre procédure dans le main.

Exercice 6 : Tri par insertion du tableau de nombres complexes

Définissez une procédure globale (pas une procédure membre) `trierParInsertion` qui prend en paramètres un tableau de nombres complexes et sa taille, et qui le trie du nombre le plus petit au plus grand (en termes de module), en utilisant l'algorithme de tri par insertion. Testez votre procédure dans le main.

TP4 : Fichiers, complexité et tris fusion

Le but de ce TP est de comparer les temps d'exécution de différents algorithmes de tri, lorsqu'on les fait tourner sur des volumes de données relativement grands. Les données à trier seront ici des nombres complexes, stockés dans un fichier. Récupérez sur le site de l'UE les fichiers `random.txt`, `sorted.txt` et `reverse.txt`.

Exercice 1 : Préambule : surcharge d'opérateurs

Afin de faciliter l'écriture des instructions utiles pour les entrées/sorties sur fichier et console, et les tris, vous allez surcharger certains opérateurs de la classe `NbComplexe` que vous avez créé au TP précédent. Reprenez votre fichier `NbComplexe.cpp` et ajoutez les opérateurs suivant : `=` (affectation), `<` (strictement inférieur, qui remplace la fonction membre `estPlusPetit`), `*` (multiplication, qui remplace la fonction membre `multiplier`), `<<` (écriture sur un flux, qui remplace la fonction membre d'affichage) et `>>` (lecture sur un flux, qui remplace la fonction membre de saisie). Vous conserverez à l'identique la fonction membre `module`, les constructeurs et le destructeur. Mettez à jour les appels dans les fonctions globales `trierParSelection` et `trierParInsertion` afin d'utiliser ces opérateurs au lieu des anciennes fonctions membres. Testez les opérateurs et les fonctions de tri.

Exercice 2 : Lire un fichier texte de nombres complexes

Ouvrez les fichiers `.txt` fournis pour en comprendre la structure, puis fermez-les. Ajoutez ensuite dans le fichier `NbComplexe.cpp` une fonction globale pour lire le contenu d'un fichier et remplir un tableau avec les nombres complexes contenus dans le fichier. Ensuite le programme triera le tableau avec l'algorithme de votre choix et l'affichera à l'écran (cf. exercices suivants). Attention, votre programme devra s'adapter automatiquement (i.e. sans qu'il y ait besoin de recompiler) au nombre de complexes contenus dans le fichier.

L'entête de cette procédure de lecture du fichier est la suivante :

Procédure `lireTabNbComplexeDepuisFichier` (`tab` : tableau de nombres complexes, `taille` : entier, `nom_fichier` : chaîne de caractères)

Précondition : `tab` n'est pas alloué, la procédure est en charge d'allouer la mémoire pour contenir les nombres complexes lus depuis le fichier. Le fichier est au format texte et commence par une ligne contenant le nombre de complexes à lire.

Postcondition : `tab` contient les nombres complexes lus depuis le fichier de nom `nom_fichier`

Paramètres en mode donnée : `nom_fichier`

Paramètres en mode donnée-résultat : `tab`, `taille`

Testez votre procédure en construisant un tableau de nombres complexes depuis l'un des trois fichiers fournis, puis affichez-le à l'écran.

Exercice 3 : Ecrire un fichier texte de nombres complexes

Ajoutez dans votre programme une procédure qui écrit un tableau de nombres complexes dans un fichier `txt`, en respectant le format des fichiers fournis sur le site de l'UE. Testez votre procédure en l'appelant sur le tableau après un tri par insertion ou par sélection.

L'entête de cette procédure d'écriture du fichier est la suivante :

Procédure `ecrireTabNbComplexeDansFichier` (`tab` : tableau de nombres complexes, `taille` : entier, `nom_fichier` : chaîne de caractères)

Précondition : `tab` contient les nombres complexes à écrire dans le fichier. Le fichier est au format texte et commence par une ligne contenant le nombre de complexes du fichier.

Postcondition : le fichier de nom `nom_fichier` contient les nombres complexes du tableau `tab`

Paramètres en mode donnée : `tab`, `taille`, `nom_fichier`

Exercice 4 : Mesurer les temps d'exécution des algorithmes de tri

Dans cet exercice vous allez tester le comportement du tri par sélection et du tri par insertion en exécutant votre programme sur les trois fichiers txt fournis. Notez à chaque fois le temps d'exécution pour compléter le tableau suivant. Ce tableau sera augmenté par les colonnes pour le tri par fusion interne et le tri par fusion externe dans les deux exercices suivants. Vous pouvez vous reporter à l'annexe C pour apprendre comment mesurer le temps d'exécution d'un bout de code.

Fichier à trier	Tri par sélection	Tri par insertion
random.txt		
sorted.txt		
reverse.txt		

Lequel des deux algorithmes est le plus performant sur un fichier aléatoire ? Même question pour un fichier déjà trié et un fichier trié dans l'ordre inverse ?

Exercice 5 : Implémenter et mesurer le temps d'exécution du tri par fusion interne

Implémentez le tri par fusion interne vu en TD (TD4 - exercice 8). Comme pour le tri par sélection ou par insertion, les données du fichier sont entièrement chargées dans un tableau, puis on trie le tableau en fusionnant des monotonies de taille 1, puis de taille 2, puis de taille 4, etc. Mesurez ensuite les temps d'exécution de cet algorithme sur les trois fichiers fournis et comparez-les aux temps obtenus pour les deux autres tris.

Exercice 6 : Implémenter et mesurer le temps d'exécution du tri par fusion externe

Les algorithmes de tri interne ne sont pas adaptés pour de très gros volumes de données. Dans ce cas, il est préférable d'utiliser des algorithmes de tri externe, qui ne chargent pas l'ensemble des données en mémoire, et qui accèdent séquentiellement aux données. Dans cet exercice, vous allez implémenter le tri externe par fusion, tel que vous l'avez vu en CM et en TD (TD5 - exercices 1 et 2). Contrairement à ce que vous avez fait pour les autres algorithmes, vous n'allez pas charger l'ensemble du fichier dans un tableau. La ligne d'entête du fichier devient donc inutile. Codez les procédures `eclatement` (TD5 - exercice 2.e) et `fusion` (TD5 - exercice 1), et `triParFusionExterne` (TD5 - exercice 2.d). Mesurez ensuite les temps d'exécution de cet algorithme sur les trois fichiers fournis et comparez-les aux temps obtenus pour les trois autres tris.

TP5 : Tableau dynamique

Dans ce TP, vous allez implémenter en C++ un module `TableauDynamique` qui devra être utilisable par d'autres. Dans cette implémentation, les cases du tableau seront numérotées à partir de 0. Vous allez créer votre classe sous forme de type de données abstrait, c'est-à-dire avec l'interface (fichier .h) séparée de l'implémentation (fichier .cpp). Ce sera aussi l'occasion pour vous d'écrire votre premier Makefile et de vous entraîner à bien valider vos procédures et fonctions membres au fur et à mesure.

Exercice 1 : Initialisation et testament, première compilation multi-fichiers

- Le type `TableauDynamique` doit permettre de stocker des éléments de type « `ElementTD` ». Vous trouverez les fichiers `ElementTD.h`, `ElementTD.cpp` et `TableauDynamique.h` sur le site de l'UE. Dans votre répertoire LIFAP3, créez un sous-répertoire TP5 et enregistrez ces fichiers dedans.
- Examinez le contenu du fichier `TableauDynamique.h` : il contient les prototypes (= déclarations) des fonctions et procédures membres promis par le module.
- Créez un nouveau fichier « `TableauDynamique.cpp` », destiné à contenir l'implémentation de votre module. Dans ce fichier, commencez par indiquer que vous allez utiliser les types et sous-programmes déclarés dans les fichiers `TableauDynamique.h` et `ElementTD.h`, grâce à la directive de précompilation `#include`. Ecrivez ensuite la **définition** du constructeur par défaut (initialisation à une case vide), et celle du destructeur (libération de la mémoire allouée sur le tas) en respectant bien les prototypes déclarés dans le .h. Enregistrez votre fichier mais ne le fermez pas.
- Créez un nouveau fichier « `main.cpp` » et écrivez un programme principal qui appelle les procédures que vous venez d'écrire, afin de tester si elles fonctionnent bien. Enregistrez votre fichier mais ne le fermez pas.
- Compilez votre programme en tapant successivement les commandes suivantes (ne passez pas à la commande 2 tant que vous avez des erreurs sur la commande 1, etc) :

```
g++ -Wall -c ElementTD.c
g++ -Wall -c TableauDynamique.c
g++ -Wall -c main.c
g++ TableauDynamique.o main.o ElementTD.o -o monprog.out
```

Notez bien l'option `-c` dans les 3 premières commandes : elle indique à gcc de s'arrêter à l'étape de traduction de votre code en langage binaire, sans essayer de créer un exécutable. C'est la 4^e commande qui crée l'exécutable en éditant les liens entre les 3 fichiers .o et avec les bibliothèques (ex. `iostream`).

Exercice 2 : Création d'un fichier Makefile et utilisation de la commande make

Créez un nouveau fichier Makefile (sans extension), qui permettra de compiler automatiquement le programme, en tapant simplement `make` au lieu des 4 commandes précédentes. Inspirez vous de l'exemple vu en cours magistral.

Exercice 3 : Implémentation des fonctionnalités de base du module

Définissez dans `TableauDynamique.cpp` les fonctionnalités ci-dessous, en respectant les prototypes déclarés dans le .h. Remarque importante : Testez *au fur et à mesure* que vous ajoutez une fonctionnalité : appelez-la dans le main, enregistrez tous vos fichiers, recompilez en utilisant la commande `make` et exécutez le programme.

- ajout d'un élément en fin de tableau,
- accès à l'élément de la case `i`,

- c. modification de l'élément de la case i ,
- d. affichage du tableau,
- e. suppression d'un élément.

Exercice 4 : Expérimenter la complexité du remplissage du tableau

- a. Le but de cet exercice est de vérifier que le coût amorti d'un ajout dans un `TableauDynamique` est de l'ordre de 3 affectations. Pour cela, vous allez comparer le temps T_1 (en secondes) nécessaire pour faire n ajouts dans un `TableauDynamique`, avec le temps T_0 nécessaire pour faire n ajouts dans un tableau « simple », qui ne change pas de taille au fur et à mesure des ajouts (ce tableau statique simple doit donc être déjà de taille n au départ). Prenez $n = 500000$ pour cet exercice.

Rappel : Pour mesurer le temps d'exécution, référez-vous à l'annexe C.

- b. Mesurez les deux temps d'exécution pour n allant de 100 000 à 20 000 000 par pas de 100 000. En utilisant un tableur, tracez les deux courbes $T_0(n)$ et $T_1(n)$. Retrouvez-vous bien un coût 3 fois plus important pour le remplissage du tableau extensible comparé au tableau simple ?
- c. Listez les avantages et inconvénients d'un `TableauDynamique` (extensible en fonction des besoins) comparé à un tableau simple (dont la taille ne change pas).

Exercice 5 : Fonctionnalités plus avancées

- a. Testez dans votre main ce qu'il se passe lorsqu'on fait $a = b$ quand a et b sont deux tableaux dynamiques, et une libération de la mémoire de a ensuite. Expliquez. Implémentez alors le constructeur par copie.
- b. Implémentez la procédure membre d'insertion d'un élément en i -ème position.
- c. Implémentez la procédure membre de tri du tableau dynamique. Vous pouvez commencer par un tri par insertion ou sélection, puis si vous avez le temps, optez pour un tri par fusion interne ou externe (voir TP4).
- d. Implémentez la procédure de recherche d'un élément dans le tableau trié. Vous pouvez commencer par une recherche linéaire, puis si vous avez le temps, optez pour une recherche dichotomique.

TP6 : Liste doublement chaînée

Commencez par créer un répertoire TP6 à l'intérieur de votre répertoire LIFAP3. L'objectif de ce TP est d'écrire une nouvelle implémentation de liste chaînée, différente de celle vue en cours et en TD, mais ayant la même interface (mêmes services proposés aux utilisateurs du module). L'implémentation que vous allez écrire est celle d'une **liste doublement chaînée**, dans laquelle :

- chaque cellule contient un pointeur sur la cellule suivante et un pointeur sur la cellule précédente,
- la classe Liste contient un pointeur sur la première cellule et un pointeur sur la dernière cellule.

Ainsi, il est possible de parcourir la liste dans les deux sens, et d'ajouter un élément en queue de liste en temps constant.

Exercice 1 : Initialisation et Makefile

- Récupérez sur le site de l'UE les fichiers ElementL.h, ElementL.cpp et Liste.h. Enregistrez-les dans votre répertoire TP6. Toujours dans ce répertoire, créez avec gedit un nouveau fichier que vous appellerez ListeDC.cpp (DC pour doublement chaînée). Ecrivez-y les `#include` requis, puis le code du constructeur et du destructeur. Enregistrez le fichier mais ne le fermez pas, vous y reviendrez plus tard.
- Créez un nouveau fichier main.cpp, et écrivez-y un programme principal minimal, qui teste la création d'une instance de la classe Liste.
- Créez un fichier Makefile et écrivez-y les lignes nécessaires pour compiler votre programme. Inspirez-vous de ce que vous avez fait lors du TP précédent. Dans le terminal, compilez votre code (`make`), exécutez-le et corrigez-le si nécessaire.

Exercice 2 : Implémentation des fonctionnalités de base du module

Définissez dans ListeDC.cpp les fonctionnalités ci-dessous, en respectant les prototypes déclarés dans le .h. Remarque importante : Testez *au fur et à mesure* que vous ajoutez une fonctionnalité : appelez-la dans le main, enregistrez tous vos fichiers, recompilez en utilisant la commande `make` et exécutez le programme.

- ajout d'un élément en tête de liste,
- ajout d'un élément en queue de liste,
- test si la liste est vide,
- vider la liste,
- renvoie du nombre d'éléments,
- accès au i-ème élément,
- modification du i-ème élément,
- affichage de la liste (de droite à gauche, et de gauche à droite),
- suppression de l'élément de tête,
- suppression de tous les éléments.

Exercice 3 : Fonctionnalités plus avancées

- a. Testez dans votre main ce qu'il se passe lorsqu'on fait $a = b$ quand a et b sont deux listes chaînées, et qu'une liste est libérée de la mémoire ensuite. Expliquez. Implémentez alors l'opérateur d'affectation qui recopie le contenu d'une liste dans l'instance. Le contenu précédent de la liste doit être libéré.
- b. Implémentez la procédure de recherche d'un élément dans une liste quelconque.
- c. Implémentez la procédure d'insertion d'un élément en i -ème position.
- d. Implémentez la procédure de tri d'une liste doublement chaînée, en utilisant l'algorithme de tri par insertion.

TP7 : Piles et Files

Les objectifs de ce TP sont les suivants :

- Faire fonctionner ensemble différentes structures de données dynamiques dans un même programme,
- Savoir écrire un Makefile avec plus d'unités de compilation,
- Mettre en œuvre la notion d'abstraction à travers l'implémentation des services fondamentaux offerts par des modules Pile et File.

Dans ce TP, vous allez écrire un module Pile (basé sur le module TableauDynamique du TP5), et un module File (basé sur le module Liste du TP6). Vous devez avoir fini l'exercice 3 du TP5 et l'exercice 2 du TP6 de sorte à obtenir des modules TableauDynamique et Liste avec toutes les fonctionnalités de base, et correctement testés.

Exercice 1 : Classe File

- Dans votre répertoire LIFAP3, créez un répertoire TP7. Placez-y les fichiers ElementL.h et ElementL.cpp que vous trouverez sur le site de l'UE. Observez le contenu de ces fichiers : au lieu de stocker des entiers dans les listes, nous allons stocker des adresses (pointeur générique : `void *`).
- Copiez dans votre répertoire TP7 vos fichiers Liste.h et ListeDC.cpp du TP6.
- Toujours dans votre répertoire TP7, ajoutez le fichier File.h qui se trouve sur le site de l'UE. Ouvrez-le : vous verrez que la classe File contient simplement une donnée membre de type Liste et les opérations spécifiques à File : enfiler, défiler, etc. Créez un nouveau fichier File.cpp et écrivez le code des fonctions membres annoncées dans File.h. **Chacune de ces fonctions s'écrit en une ligne uniquement.**

C'est un exemple d'abstraction et d'encapsulation : la « sur-couche » File, par dessus le module Liste, permet à l'utilisateur du module File de la voir comme une boîte noire, simple à utiliser, qui ne propose que les services permis sur une File. L'utilisateur du module File n'a pas à savoir si elle est implémentée sous forme d'une liste chaînée ou d'un tableau dynamique ou autre. Idéalement, il ne voit que les services proposés dans le fichier File.h et ne risque pas d'effectuer des opérations illégales sur une File, comme une insertion en plein milieu par exemple.

- Placez dans votre répertoire TP7 une copie du Makefile de votre TP6 et complétez-le pour prendre en compte les deux unités de compilation (Liste et File). Attention à bien mettre à jour la liste des .h dans les listes de dépendances : on rappelle que pour une ligne commençant par 'fichier.o', il faut indiquer le fichier .cpp correspondant et tous les .h inclus dans ce .cpp (et seulement ceux-là). Ecrire un programme principal qui teste toutes les fonctions de la classe File, compilez et corrigez votre code jusqu'à ce qu'il fonctionne.

Exercice 2 : Classe Pile

- Copiez dans votre répertoire TP7 vos fichiers TableauDynamique.h et TableauDynamique.cpp du TP5.
- Placez dans votre répertoire TP7 les fichiers ElementTD.h, ElementTD.cpp et Pile.h que vous trouverez sur le site de l'UE. Vous observerez que le type Pile contient simplement une donnée membre de type TableauDynamique et les opérations spécifiques à Pile : empiler, dépiler, etc. Créez un nouveau fichier Pile.cpp et écrivez le code des fonctions membres annoncées dans Pile.h. **Chacune de ces fonctions s'écrit en une ligne uniquement.**
- Complétez le Makefile du TP5 pour prendre en compte les deux unités de compilation (TableauDynamique et Pile). Ecrire un programme principal qui teste toutes les fonctions de la classe Pile, compilez et corrigez votre code jusqu'à ce qu'il fonctionne.

TP8 : Arbre binaire de recherche

Les objectifs de ce TP sont les suivants :

- Etre capable d'implémenter les services fondamentaux offerts par un module Arbre Binaire de Recherche,
- Etre capable de mesurer la performance d'un arbre binaire de recherche,
- Comprendre l'influence de la hauteur de l'arbre sur la performance,
- Etre capable d'utiliser des piles et des files pour parcourir itérativement un arbre.

Exercice 1 : Classe Arbre

- Dans votre répertoire LIFAP3, créez un répertoire TP8. Placez-y les fichiers ElementA.h, ElementA.cpp et Arbre.h que vous trouverez sur le site de l'UE. Créez un nouveau fichier main.cpp et écrivez-y un programme principal vide pour l'instant. Créez également le Makefile qui permettra de compiler votre projet.
- Créez le fichier Arbre.cpp en y plaçant les définitions des fonctions membres suivantes :
 - constructeur et destructeur
 - insererElement
 - afficherParcoursInfixe
 - rechercherElement
 - vider et estVide

Remarque : Pour certaines de ces fonctions membres, vous devrez utiliser une fonction ou procédure auxiliaire, souvent récursive, travaillant sur un sous-arbre, et prenant donc comme paramètre l'adresse du nœud dans lequel le sous-arbre est enraciné. Ces fonctions auxiliaires restent internes au module Arbre : elles apparaissent donc comme privées dans le .h.

- Codez ensuite les fonctions membres hauteurArbre et profondeurMoyenneArbre.
- Dans le fichier main.cpp, écrivez un programme principal qui insère dans un arbre binaire de recherche 255 entiers aléatoires compris entre 1 et 100 000, puis qui calcule la hauteur maximale de l'arbre et sa profondeur moyenne. Testez que tout fonctionne bien, en utilisant notamment la procédure d'affichage d'arbre. Vérifiez que vous n'obtenez pas le même arbre si vous exécutez deux fois le programme.

Rappel : Pour tirer aléatoirement des valeurs, référez-vous à l'annexe B.

- Complétez le programme principal pour qu'il recherche 100 nombres aléatoires entre 0 et 100 000 dans l'arbre (bien évidemment, parmi ces 100 nombres, certains seront effectivement présents dans l'arbre et d'autres non). En plus de la hauteur maximale et de la profondeur moyenne de l'arbre, le programme devra afficher le nombre moyen de nœuds visités par opération de recherche.
- Modifiez le programme principal pour qu'il répète 60 fois le processus complet de création d'arbre + recherche de 100 éléments. Vous devez obtenir comme affichage 60 lignes, avec sur chaque ligne la hauteur maximale et le nombre moyen de nœuds visités par opération de recherche (n'affichez plus les arbres eux-mêmes). Quelle relation constatez-vous entre les deux quantités ?

Cet exercice devrait vous avoir convaincu de l'intérêt d'équilibrer les arbres. Les algorithmes d'équilibrage sont au programme de LIFAP6.

Exercice 2 : Parcours en largeur (version itérative utilisant une File)

- a. Copiez dans votre répertoire TP8 vos fichiers ElementL.h, ElementL.cpp, Liste.h, ListeDC.cpp, File.h et File.cpp du TP7.
- b. Dans Arbre.h, ajoutez la déclaration d'une procédure d'affichage **en largeur**. Dans Arbre.cpp, écrivez la définition de cette procédure. Ce code utilisera bien sûr une variable locale de type File. Vous devrez donc ajouter un #include "File.h" au début du fichier Arbre.cpp.
- c. Modifiez le programme principal pour qu'il affiche deux fois un arbre rempli : d'abord en utilisant la procédure d'affichage dans l'ordre croissant (parcours infixe), puis en utilisant votre parcours en largeur.

Exercice 3 : Parcours postfixe (version itérative utilisant deux piles)

- a. Copiez dans votre répertoire TP8 vos fichiers ElementTD.h, ElementTD.cpp, TableauDynamique.h, TableauDynamique.cpp, Pile.h et Pile.cpp du TP7.
- b. Dans Arbre.h, ajoutez la déclaration d'une procédure **itérative** d'affichage **postfixe**. Dans Arbre.cpp, écrivez la définition de cette procédure avec l'algorithme suivant, qui utilise deux piles d'adresses de nœuds, qu'on appellera pileA et pileB :
 - on place la racine dans la pileB,
 - tant qu'on n'a pas épuisé la pileB, on déplace le sommet de la pileB vers la pileA, et on empile son fils gauche (s'il existe) puis son fils droit (s'il existe) dans la pileB, finTantQue
 - une fois la pileB vide, on dépile la pileA jusqu'à la vider, en affichant au fur et à mesure les éléments contenus dans les nœuds.
- c. Modifiez le programme principal pour qu'il affiche aussi le contenu de l'arbre par ce troisième parcours.

Annexes

Notes :

[illegible]

Annexe A : Commandes Linux usuelles

Action	Commande
Obtenir de l'aide sur une commande	<code>man commande</code>
Chercher les commandes relatives à un mot-clé	<code>man -k motcle</code>
Obtenir des informations sur l'utilisateur courant	<code>who am i</code>
Lister le contenu du répertoire courant	<code>ls</code> <code>ls -a</code> (affiche aussi les fichiers cachés) <code>ls -al</code> (affichage détaillé)
Lister le contenu d'un répertoire autre que le répertoire courant	<code>ls cheminrepertoire</code> <code>ls -a cheminrepertoire</code> <code>ls -al cheminrepertoire</code>
Changer de répertoire	<code>cd cheminrepertoire</code>
Aller au répertoire père	<code>cd ..</code>
Aller à la racine de son répertoire personnel	<code>cd</code> <code>cd ~</code>
Afficher le chemin du répertoire courant	<code>pwd</code> (print working directory)
Créer un répertoire	<code>mkdir cheminrepertoire</code>
Visualiser le contenu d'un fichier texte	<code>more cheminfichier</code> (entree = ligne suivante ; espace = page suivante ; q = quitter)
Editer un fichier texte	<code>gedit cheminfichier &</code>
Faire une copie d'un fichier	<code>cp cheminsource chemincible</code>
Déplacer ou renommer un fichier	<code>mv cheminsource chemincible</code>
Supprimer un fichier	<code>rm cheminfichier</code>
Supprimer un répertoire et tout son contenu	<code>rm -r cheminrepertoire</code>

Notion de chemin

Le chemin permet de savoir où se trouve un fichier ou un répertoire dans l'arborescence. Deux types de chemins sont utilisés sous Linux :

- les chemins absolus : ils indiquent tout le chemin d'accès à partir de la racine du système (/)
- les chemins relatifs : ils indiquent le chemin à partir du point où l'on est dans l'arborescence (répertoire courant). Le chemin relatif permettant d'accéder au répertoire père du noeud courant est ..

Exemples :

- `/home/b/p0123456/LIFAP3/TP1` est le chemin absolu du répertoire TP1.
- `/home/b/p0123456/LIFAP3/TP2` est le chemin absolu du répertoire TP2.
- `/home/b/p0123456/LIFAP3/TP1/hello.cpp` est le chemin absolu du fichier hello.cpp.
- Si on est dans le répertoire LIFAP3, alors le chemin relatif d'accès au répertoire TP1 est simplement TP1 (ou `./TP1`).
- Si on est dans le répertoire TP1, alors :
 - le chemin relatif d'accès au répertoire LIFAP3 est `..`
 - le chemin relatif d'accès au répertoire TP2 est `../TP2`
 - le chemin relatif d'accès au fichier hello.cpp est `hello.cpp` (ou `./hello.cpp`)

Lorsque vous vous connectez, vous êtes placé dans votre répertoire personnel (par exemple `/home/b/p0123456`).

Annexe B : Tirage de nombres aléatoires

Les ordinateurs étant des machines déterministes, leur demander de produire des nombres réellement aléatoires n'est pas du tout trivial. Il existe plusieurs solutions de différentes qualités et donc de coûts différents. Pour les domaines dans lesquels le caractère réellement aléatoire est crucial, comme la cryptographie (les clefs de chiffrement doivent être parfaitement aléatoires pour garantir une sécurité maximale), on peut connecter l'ordinateur à un appareil spécifique qui génère des nombres aléatoires. Pour d'autres domaines, il est possible de se contenter de nombres dits « pseudo-aléatoires » et d'utiliser une méthode algorithmique, sans appareil autre que l'ordinateur lui-même. On utilise alors une fonction appelée générateur de nombres pseudo-aléatoires.

Un générateur de nombres pseudo-aléatoires peut être vu comme une suite avec un u_0 et une relation de récurrence $u_{n+1} = f(u_n, u_{n-1}, u_{n-2} \dots)$ suffisamment compliquée pour que les valeurs $u_1, u_2, u_3 \dots$ *semblent* sans rapport entre elles, et *semblent* donc aléatoires. En réalité, la fonction f est bien déterminée et les nombres ne sont donc pas réellement aléatoires. On dit qu'ils sont pseudo-aléatoires.

La bibliothèque **stdlib** du langage C fournit un générateur « basique » de nombres pseudo-aléatoires. Ce générateur est très loin d'être parfait, mais il suffira pour nos besoins. Les fonctions à appeler sont **srand** (appelée une seule fois au début du programme pour initialiser le générateur) et **rand** (appelée à chaque fois que l'on a besoin d'un nombre aléatoire).

- La fonction `srand(unsigned int graine)` prend en paramètre un entier appelé « graine », qui va être utilisé comme u_0 pour initialiser la suite de nombres. On pourrait demander à l'utilisateur de saisir au clavier un entier de son choix et l'utiliser comme graine. Mais souvent, on préfère fabriquer automatiquement un entier à partir de l'heure précise à laquelle l'exécution est lancée, et utiliser cet entier comme graine. De cette façon, on obtiendra automatiquement des tirages différents à chaque exécution, ce qui est en général souhaitable (sauf lorsqu'on débogue...).
- La fonction `rand()` simule une loi uniforme et renvoie un entier pseudo-aléatoire compris entre 0 et `RAND_MAX`. `RAND_MAX` est une constante définie dans `stdlib.h`, sa valeur dépend des implémentations mais elle est souvent égale à 0x7FFF (en hexadécimal), ie. le plus grand entier relatif codable sur 4 octets. Pour obtenir un entier dans une plage donnée, il faudra donc renormaliser la valeur donnée par `rand()`.

Exemple :

```
#include <stdlib.h>    // pour srand() et rand()
#include <time.h>       // pour time()

int main() {
    int i, aleatoire;
    int min = 1, max = 31;
    int plage = max - min + 1 ;

    /* Initialisation du générateur, à ne faire qu'une fois dans le programme */
    srand((unsigned int)time(NULL));

    /* Tirage de 100 entiers aléatoires compris entre min et max inclus */
    for (i = 0; i < 100; i++) aleatoire = min + (int)(plage * (rand() / (RAND_MAX + 1.0)));

    return 0;
}
```


Annexe C : Mesure de temps d'exécution

Il est souvent utile de pouvoir chronométrer le temps d'exécution d'un bout de code. Il existe une fonction C permettant de récupérer le temps système, i.e. le nombre de ticks de l'horloge interne écoulés (en général depuis le démarrage du système) : la fonction `clock_t clock()` ; Cette fonction et le type `clock_t` sont définis dans la bibliothèque standard **time**.

En calculant la différence (en nombre de ticks) entre le temps système à deux instants du code, on peut donc mesurer le temps passé entre ces deux instants. Pour convertir le nombre de ticks écoulés en secondes, il existe une constante (dépendante de l'OS et de la machine) nommée `CLOCKS_PER_SEC` (aussi définie dans la bibliothèque **time**) donnant le nombre de ticks de l'horloge interne dans une seconde. Diviser le nombre de ticks mesurés par cette constante donne donc le temps, en secondes, écoulé entre les deux appels à `clock()`.

Exemple :

```
#include <iostream>    // pour cout
#include <time.h>       // pour clock
using namespace std;

void procedureAMesurer () { ... }

int main() {

    clock_t tempsExecution = clock();
    procedureAMesurer();
    tempsExecution = clock() - tempsExecution;
    cout << "Execution en " << ((float)tempsExecution)/CLOCKS_PER_SEC << " secondes.";

    return 0;
}
```

Remarques :

- La précision de cette horloge interne n'est pas très bonne (en général autour de 10^{-3} secondes) alors que l'on peut souhaiter mesurer des bouts de code bien en dessous de la milliseconde. Il existe une bibliothèque C++ avec une meilleure précision, la bibliothèque **chrono**. La classe à utilisée est alors `high_resolution_clock` incluant en particulier la fonction membre `now()` (équivalent de l'appel `clock()` de **time**).
- Il existe d'autres fonctions et méthodes de mesure, souvent OS-dépendantes, comme la commande `time` sous Linux. Pour mesurer le temps d'exécution total d'un programme tapez simplement à l'exécution du programme dans un terminal : `time ./monExecutable.out`. Attention, cette méthode ne peut pas être utilisée pour mesurer qu'une partie du code.