LIFAPSD : Algorithmique, Programmation et Structures de données

Fascicule de travaux dirigés et de travaux pratiques

Nicolas Pronost Université Claude Bernard Lyon 1

Notes :

Table des matières

Travaux dirigés

Notes :

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

Exercice 1 : 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 {
    /* dessiner l'état de la mémoire */
  }
}

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;
}
```

a. Que fait la procédure `mystere` ? Autrement dit, si vous deviez lui donner un nom plus explicite, lequel choisiriez-vous ?

b. Dessinez l'état de la pile au moment indiqué en commentaire. Vous utiliserez le modèle théorique de pile vu en cours. Vous supposerez que la valeur de retour du main est stockée à l'adresse 3 987 546 988.

Exercice 2 : Tableaux de structures et fonction récursive

Soit le programme C++ suivant :

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

struct Point {
  double x;
  double y;
};
```

```
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 superieur 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;
}
```

a. 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) ?

b. Combien d'octets occupe le tableau cheminTriangle ?

c. La fonction longueurchemin est-elle récursive ? Même question pour la fonction dist.

d. A quoi sert le mot-clé const dans l'entête de la fonction longueurchemin ?

e. Dessinez l'évolution de la pile lors de l'exécution de ce programme, en utilisant le modèle théorique de pile vu en cours. Vous supposerez que la valeur de retour du main est stockée à l'adresse 3 987 546 998. Vous ferez un dessin à chaque fois que vous êtes sur le point de sortir d'un sous-programme (i.e. après l'affectation de la valeur de retour et avant le retour au programme appelant).

f. Combien de fois la fonction longueurchemin est-elle appelée ? Même question pour la fonction dist.

g. Que pensez-vous de l'implémentation récursive de longueurchemin, en termes de ressources mémoire ?

h. 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

Soit le programme C++ suivant :

```
#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;
}
```

a. Expliquez ce que signifie l'instruction `p1 = new int [5];`

b. Proposez une autre façon d'écrire l'instruction `(*p1)*=5;`

c. Faites les quatre traces mémoire de ce programme, en supposant que la valeur de retour du main est stockée à l'adresse 3 566 758 966 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 au TD précédent. Vous supposerez 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 dynamique de mémoire.

```
/* Précondition: 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 (1) */
}

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 (2) */
  delete [] somme;
  return 0;
}
```

TD3 : Classe et objet (1/2)

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.

a. Ecrivez, en C++, la classe Personne.

b. Donner le programme principal, en C++, permettant de saisir puis d'afficher les informations d'une personne.

c. En programmant la procédure membre d'affichage, vous avez dû choisir sous quel format la personne est affichée, et en particulier quel(s) caractère(s) de séparation utiliser entre les différentes données membres à afficher. Ecrire une deuxième version de cette procédure, utilisant le principe de surcharge, permettant de paramétrer le(s) caractère(s) utilisé(s).

d. Modifier cette procédure pour qu'elle prenne une valeur de paramètre par défaut. Voyez-vous un problème ?

e. Que doit-on faire pour interdire aux utilisateurs de la classe de manipuler les données membres directement ?

f. 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 ?

g. Ecrivez un programme principal qui modifie le nom d'une personne après saisie.

Exercice 2 : Surcharge d'opérateurs

On souhaite pouvoir faire des opérations mathématiques sur des points 2D. 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 additioner, 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 : Classe et objet (2/2)

Exercice 1 : 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;
}
```

a. 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.

b. 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 2 : 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() const {return sqrt(x*x+y*y);}
    float distancePoint (const Point2D & p) const {
       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 distpt1O;
  float distpt2pt1;
  distpt1O = 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.

TD5 : Algorithme, invariant de boucle et complexité (1/2)

Exercice 1 : 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 ← 0
  i ← 1
  Tant que (i < n) Faire
      j ← i + 1
      Tant que (j ≤ n) Faire
          compteur ← compteur + 1
          j ← j + 1
      Fin tantque
      i ← i * 2
  Fin tantque
  Afficher(compteur)
Fin
```

a. Quelle est la valeur finale du compteur dans le cas où n = 16 ?

b. 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.

c. Réexprimez le résultat précédent en fonction de n.

Exercice 2 : 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.

a. Dérouler le tri par insertion du tableau {5.1, 2.4, 4.9, 6.8, 1.1, 3.0}.

b. 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 : tab[1] ≤ tab[2] ≤ … ≤ tab[n]
Paramètres en mode donnée : aucun
```

Paramètre en mode donnée-résultat : tab

c. 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.

d. 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 (tri du minimum) vu en cours ?

TD6 : Algorithme, invariant de boucle et complexité (2/2)

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 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.

a. 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' ?

b. 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

c. 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 ?

d. Quelle serait l'entête de la procédure en C++ ?

e. Donner un invariant de la boucle introduite dans la procédure chiffrer.

TD7 : Tri fusion (1/2)

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) : **val1, val2, fic1, fic2, ficSortie, succeslect1, succeslect2**. Vous pouvez en ajouter d'autres si nécessaire.

Exercice 2 : Complexité du 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

a. 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.

b. 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 ?

c. 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 ($n = 2^p$) ?

TD8 : Tri fusion (2/2)

Exercice 1 : Tri fusion sur fichier

a. Ecrire en langage algorithmique la procédure de tri par fusion d'un fichier, en supposant que vous disposez déjà des procédures « eclatement » et « fusion » (voir les entêtes ci-dessous). La procédure de tri doit appeler les procédures « eclatement » et « fusion » jusqu'à ce que le fichier soit complètement trié.

Procédure eclatement (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 monotonies, 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 monotonies 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 monotonies, alors la dernière monotonie de FicB peut être incomplète.
Postconditions : le fichier appelé nomFicX contient nbMonoDansX monotonies de longueur 2*lg, la dernière pouvant être plus courte. Ces monotonies résultent de la fusion 2 à 2 des monotonies 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

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

Exercice 2 : Tri fusion interne

Ecrire une version itérative (c'est-à-dire non récursive) de l'algorithme du tri fusion d'un tableau de réels. Cette version du tri fusion n'utilisera pas de fichiers, on pourra utiliser de la mémoire sur le tas pour stocker des (sous-) tableaux.

TD9 : Tableau dynamique

Exercice 1 : 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 2 : Crible d'Eratosthène

a. On se propose de calculer tous les nombres premiers plus petits qu'un entier $n > 1$ 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 ?

b. Ecrire en C++ la procédure **eratosthene** qui calcule les nombres premiers plus petits que n passé en paramètre.

Exercice 3 : 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 ?

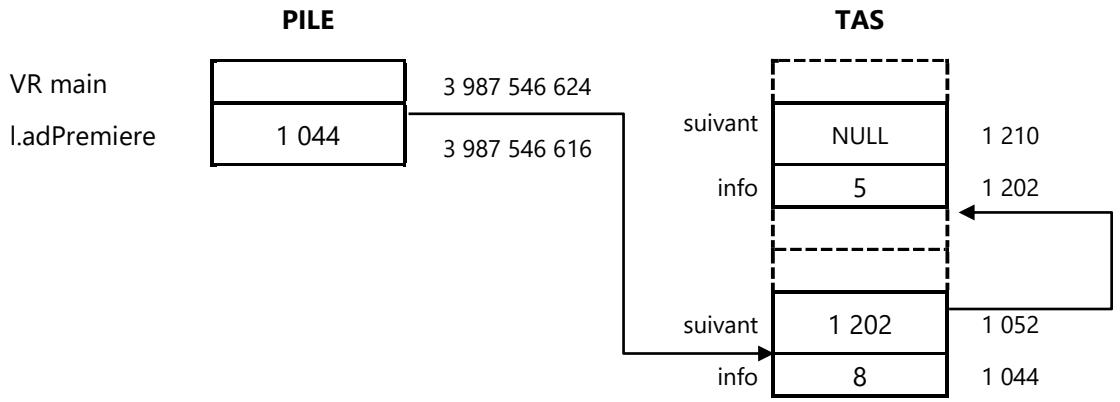
TD10 : Liste chaînée

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

Exercice 1 : Dynamique des données en mémoire

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

```
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;
}
```



a. Utilisez la représentation graphique présentée en cours pour décrire la liste créée par ce programme.

b. Complétez les pointillés des appels `l.ajouterEn.......(8)` et `l.ajouterEn.......(5)`.

c. 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 : 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 3 : Inversion d'une liste

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

Exercice 4 : Suppression des occurrences d'un élément

Ecrire une procédure membre qui supprime toutes les occurrences d’un élément e dans une liste.

TD11 : Pile et file

Exercice 1 : Validité du parenthésage d'une expression

Un problème fréquent pour les compilateurs et les traitements de texte 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 retourne vrai si l'expression passée en paramètre est valide, faux sinon.

Exercice 2 : 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.

a. Créer une fonction `estOperateur` en C++ qui prend une chaîne de caractères en paramètre (classe string) et retourne vrai si cette chaîne représente un opérateur valide (ici seulement "+", "-", "/" ou "*"), et faux sinon.

b. Créer une fonction `calcul` qui prend en paramètre une chaîne de caractères représentant un opérateur `op` et deux opérandes réelles `a` et `b`, et qui retourne la valeur numérique `a op b`.

c. En utilisant les deux fonctions précédentes, écrire la fonction `polonaise`. Vous pourrez convertir une chaîne de caractères en un réel en utilisant les instructions suivantes :

```
double monReel = std::stod(maChaineRepresentantUnReel);
```

TD12 : Arbre binaire

Exercice 1 : Parcours préfixé itératif d'un arbre binaire

Ecrire en langage C++ la procédure membre **itérative** du parcours **préfixé** dans un arbre binaire qui affiche les éléments de l'arbre.

Exercice 2 : Suppression dans un ABR

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

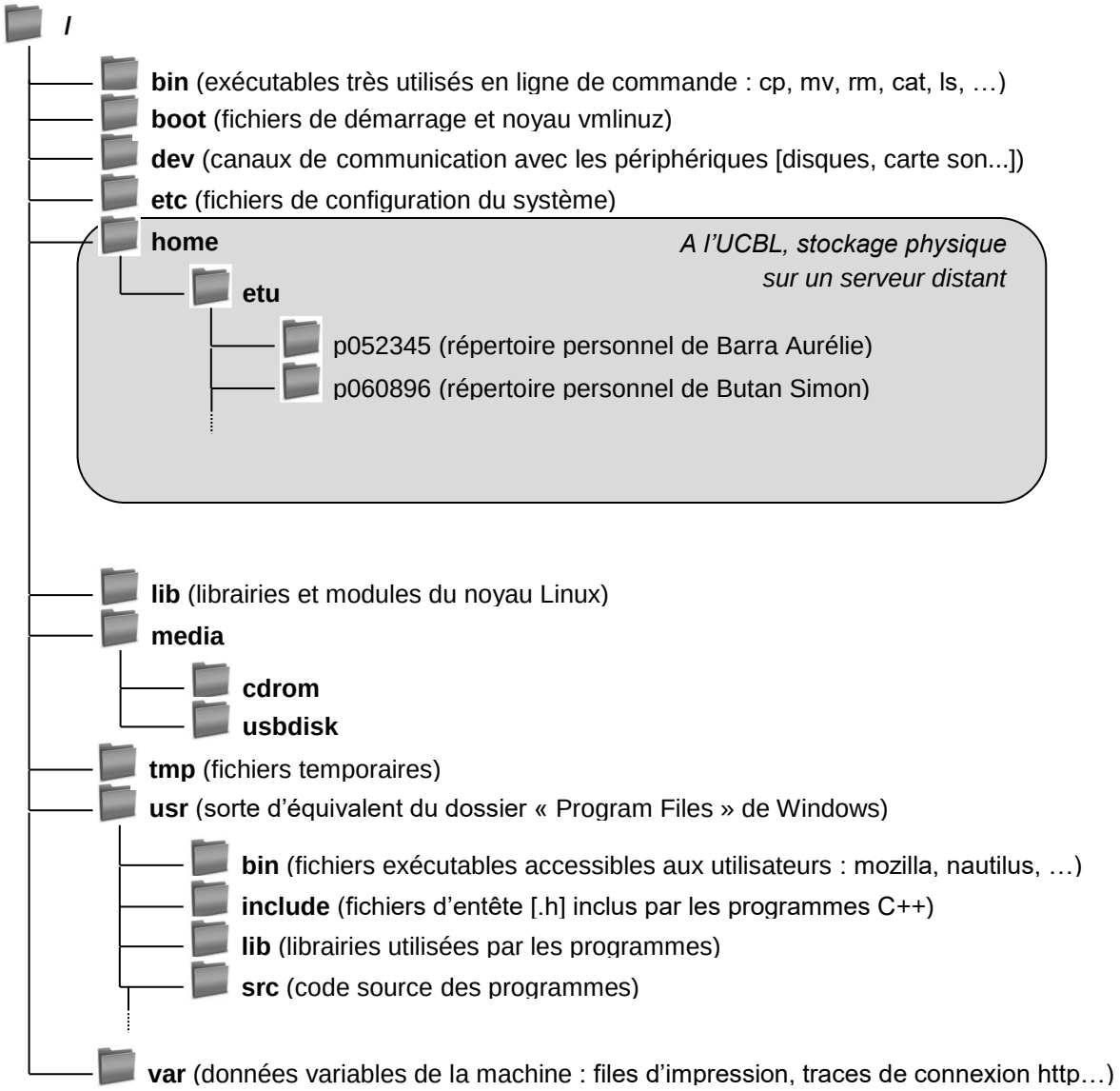
Travaux pratiques

Notes :

TP1 : De LIFAPI à LIFAPSD…

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.



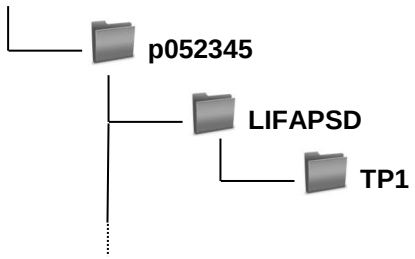
A l'université, les répertoires des utilisateurs ne sont pas stockés physiquement sur les machines des salles de TP, mais sur un serveur auquel les machines de TP accèdent par le réseau (même principe que pour votre lecteur W: sous Windows). Mais cela est transparent pour l'utilisateur, qui accède toujours à son répertoire personnel en allant dans `/home/etu/[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 LIFAPSD en tapant la commande suivante (respectez bien l'espace après mkdir, mais n'en mettez pas dans le nom du répertoire) : `mkdir LIFAPSD`
- vérifiez que ce nouveau répertoire apparaît dans le répertoire courant, en tapant `ls`
- allez dans le répertoire créé en tapant `cd LIFAPSD`
- 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 LIFAPI, 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 utilisez 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 ~/LIFAPSD/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 !" << endl;
  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.

a. Dans le fichier `Vecteur3D.cpp`, définisser une structure `Vecteur3D` qui contient trois champs de réels (x, y et z).
b. Puis ajouter les procédures et fonctions suivantes
   - une fonction `Vecteur3DGetNorme` qui retourne la norme d'un vecteur passé en paramètre. Pour rappel la norme (ou taille) d'un vecteur est donnée par la formule : $\sqrt{x^2 + y^2 + z^2}$.
   - une procédure `Vecteur3DNormaliser` qui normalise le vecteur passé en paramètre. Normaliser met à l'échelle le vecteur de telle sorte que sa norme vaut 1.

- 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: ";
   Vecteur3DAfficher(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 :

a. 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 -10.0 et 10.0 (avec un chiffre après la virgule). La taille du tableau est aussi passée en paramètre. Consulter l'annexe B pour apprendre comment générer aléatoirement des valeurs.

b. 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.
c. 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 créé 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.

Exercice 5 : Manipulation de tableaux de structures

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

a. Une procédure `Vecteur3DConcatenationTabVecteurs` qui prend trois tableaux de Vecteur3D en paramètre, ainsi que la taille des deux premiers. Cette procédure remplit le troisième tableau déjà alloué en mémoire et de la bonne taille avec les vecteurs contenus dans le premier et le second tableau, dans cet ordre.
b. Une procédure `Vecteur3DInverseTabVecteurs` qui inverse le contenu d'un tableau de Vecteur3D passé en paramètre, avec sa taille.

Ecrire le programme principal qui créé deux tableaux respectivement de 5 et 6 vecteurs aux valeurs réelles aléatoires, les affiche, puis affiche le tableau issu de la concaténation des deux tableaux. Ajouter ensuite les instructions permettant d'inverser le tableau concaténé puis de l'afficher. Compiler et exécuter votre programme, et vérifier que vos procédures 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 :

a. 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 ?

b. Dans quel ordre sont empilés les paramètres d'une fonction ou d'une procédure ?

c. Dans quel ordre sont empilées les données membres d'un objet ?

d. Les variables locales d'un même sous-programme sont-elles, comme en TD, dans l'ordre dans lequel elles étaient déclarées dans le code ?

e. 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 ?

f. 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

a. Tapez le programme ci-dessous qui calcule le nombre de combinaisons de p parmi n dans un nouveau fichier. Compilez-le et exécutez-le pour vérifier que tout fonctionne bien.

```
1  #include <iostream>
2  using namespace std;
3
4  /* Calcul d'un coefficient binomial à l'aide du triangle de Pascal */
5  int comb(int n, int p) {
6    int tmp1, tmp2;
7    cout <<"Calcul du nb de combinaisons de "<<p<<" elts parmi "<<n<<endl;
8
9    if ((p==0) || (n==p))
10     return 1;
11   tmp1 = comb(n-1, p-1);  /* premier appel récursif */
12   tmp2 = comb(n-1, p);    /* second appel récursif */
13   return tmp1 + tmp2;
14 }
15
16 int main() {
17   int c;
18   c = comb(4, 3);
19   cout << "c vaut " << c << endl;
20   return 0;
```

```
21 }
```

b. Puis recompilez-le en ajoutant à GCC l'option `-g` (cette option ajoute à l'exécutable des informations de débogage), et lancez le débugger gdb avec la commande `gdb nomDeVotreExecutable`. 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.

c. 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 ?

d. 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 ?

e. 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. 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.

a. Dans le fichier écrit à l'exercice 2, indiquez les pré- et post-conditions et résultat en commentaires de la fonction déjà réalisée.

b. 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 précédement et vérifier que vous obtenez le même résultat (l'afficher).

c. 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 précédement et vérifier que vous obtenez le même résultat (l'afficher).

Exercice 5 : Arithmétiques des pointeurs

Ecrivez un programme principal exécutant les instructions suivantes :

- Afficher la taille d'un Vecteur3D en mémoire et la taille d'un pointeur sur Vecteur3D.
- Allouer un tableau `tabVecteurPile` de 3 Vecteur3D sur la pile et un autre tableau `tabVecteurTas` de même taille sur le tas.
- Remplir les deux tableaux avec les mêmes valeurs, tirées aléatoirement entre -10.0 et 10.0 avec exactement un chiffre après la virgule.
- Afficher la taille des variables `tabVecteurPile` et `tabVecteurTas`.
- Afficher l'adresse du premier élément des deux tableaux.
- Afficher les adresses des champs x,y,z du deuxième élément des deux tableaux.
- Afficher le contenu de `*(tabVecteurPile+2)` et `*(tabVecteurTas+2)`. A quoi cela correspond-t-il ?
- Affecter au champ x du premier élément de `tabVecteurPile` la différence entre l'adresse du champ y du deuxième élément de `tabVecteurTas` et l'adresse du champ z du troisième élément de `tabVecteurTas`. Faire une version où les adresses sont converties en `long int` avant soustraction, et une version où elles ne le sont pas. Afficher et expliquer les valeurs obtenues.

Exercice 6 : Pointeurs et références

Soient les trois instructions :

```
int x = 1;
int & rx = x;
int * px = &x;
```

a. Afficher la valeur de x, la valeur de rx et la valeur de px.
b. Afficher l'adresse mémoire de la variable x, celle de rx et celle de px. Expliquer ces valeurs.
c. Ecrire trois procédures `procedureAvecPointeur (int * ptr)`, `procedureAvecReference (int & rf)` et `procedureAvecInt (int val)` qui affichent la valeur et l'adresse du paramètre.
d. Appeler ces trois procédures avec comme paramètre px pour la première, x puis rx pour la deuxième et la troisième. Expliquer les résultats affichés.

TP3 : Classe et objet

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

Exercice 1 : Définition du type nombre complexe

a. 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`).

b. Ajoutez deux procédures 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.

c. Créez un main qui crée un nombre complexe sur la pile, l'affiche, puis le saisi et le réaffiche. Qu'obtenez-vous ?

d. Quelle instruction permet d'afficher la taille, en octets, d'un nombre complexe ?

Exercice 2 : Constructeur, destructeur et allocation dynamique

a. 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.

b. 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.

Rappel : $(re_1 + im_1\, i) \times (re_2 + im_2\, i) = (re_1 \times re_2 - im_1 \times im_2) + (im_1 \times re_2 + re_1 \times im_2)\, i$

c. 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. Finallement, 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

a. 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 exactement 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 remplit 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 : Fichier et complexité expérimentale

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 : surchage 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` (ou celui du corrigé) et ajoutez les opérateurs suivants : = (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. 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. N'oubliez pas d'inclure la librairie `fstream` lors de vos tests.

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.
Postcondition : le fichier de nom nom_fichier est au format texte et commence par une ligne contenant le nombre de complexes du fichier, et il contient ensuite 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. Vous pouvez vous reportez 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 ?

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 de leur implémentation.

Exercice 1 : Création et destruction, première compilation multi-fichiers

a. 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 LIFAPSD, créez un sous-répertoire TP5 et enregistrez ces fichiers dedans.

b. Examinez le contenu du fichier TableauDynamique.h : il contient les prototypes (= déclarations) des fonctions et procédures membres promis par le module.

c. 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.

d. 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.

e. 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.cpp
g++ -Wall -c TableauDynamique.cpp
g++ -Wall -c main.cpp
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 4ème 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.

a. vider le tableau,

b. ajouter un élément en fin de tableau,

c. accéder à l'élément de la case i,

d. modifier l'élément de la case i,

e. afficher le tableau,

f. supprimer 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 b 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 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.

d. 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.

TP6 : Liste doublement chaînée

Commencez par créer un répertoire TP6 à l'intérieur de votre répertoire LIFAPSD. 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 tête et en queue de liste en temps constant.

Exercice 1 : Initialisation et Makefile

a. 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 Liste.cpp. Ecrivez-y les #include requis, puis le code du constructeur. Enregistrez le fichier mais ne le fermez pas, vous y reviendrez plus tard.

b. Créez un nouveau fichier main.cpp, et écrivez-y un programme principal minimal, qui teste la création d'une instance de la classe Liste.

c. 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 Liste.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.

a. affichage de la liste (de droite à gauche, et de gauche à droite),

b. test si la liste est vide,

c. ajout d'un élément en tête de liste,

d. ajout d'un élément en queue de liste,

e. suppression de l'élément de tête,

f. vider la liste,

g. le destructeur,

h. renvoie du nombre d'éléments,

i. accès au i-ème élément,

j. modification du i-ème élément.

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 : Pile et file

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

a. Dans votre répertoire LIFAPSD, 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 *`).

Cela veut donc dire que l'on insèrera un élément (par exemple un entier) en passant en paramètre de `enfiler` un pointeur sur cet élément (pouvant être soit sur la pile soit sur le tas). Ex. : `mafile.enfiler(new int(1));`

De même lorsque l'on souhaite récupérer un élément depuis la file, nous allons récupérer par défaut un `void*` que l'on va ensuite convertir dans le type souhaité. Ex. : `int * elem = (int*) mafile.premierDeLaFile();`

b. Copiez dans votre répertoire TP7 vos fichiers Liste.h et Liste.cpp du TP6.

c. 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.

d. 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

a. Copiez dans votre répertoire TP7 vos fichiers TableauDynamique.h et TableauDynamique.cpp du TP5.

b. 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**.

c. 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

a. Dans votre répertoire LIFAPSD, 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.

b. Créez le fichier Arbre.cpp en y plaçant les définitions des fonctions membres suivantes :

   - constructeur et destructeur
   - vider et estVide
   - insererElement
   - afficherParcoursInfixe
   - rechercherElement

   Remarque : Pour certaines de ces fonctions membres, vous devrez utiliser une fonction ou procédure auxiliaire, souvent récursive, travaillant sur un sous-arbre, et prenant 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.

c. Codez ensuite la fonction membre hauteurArbre.

d. 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 de l'arbre. 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.

e. Complétez le programme principal pour qu'il recherche 100 nombres aléatoires entre 1 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 de l'arbre, le programme devra afficher le nombre moyen de nœuds visités par opération de recherche.

f. 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 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 ?

Vous pouvez également afficher la hauteur et le nombre de nœuds visités pour des arbres qui ont été remplis avec des tableaux triés (produisant des arbres dégénérés) et des tableaux organisés afin de créeer des arbres équilibrés.

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, Liste.cpp, File.h et File.cpp du TP7.

b. Dans Arbre.h, ajoutez la déclaration d'une procédure **itérative** 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 :

Annexe A : Commandes Linux usuelles

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

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/LIFAPSD/TP1` est le chemin absolu du répertoire TP1.
- `/home/b/p0123456/LIFAPSD/TP2` est le chemin absolu du répertoire TP2.
- `/home/b/p0123456/LIFAPSD/TP1/hello.cpp` est le chemin absolu du fichier hello.cpp.
- Si on est dans le répertoire LIFAPSD, 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 LIFAPSD 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 $u_{n+1} = f(u_n, u_{n-1}, u_{n-2} \ldots)$ suffisamment compliquée pour que les valeurs u_1, u_2, u_3 … *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 convertir la valeur donnée par rand().

Exemple :

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

int main() {
  int 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 (int i = 0; i < 100; i++) aleatoire = (rand() % plage) + min;

  return 0;
}
```

Annexe C : Mesure de temps d'exécution

Il est souvent utile de pouvoir chronometrer 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.