

CORRIGES

Cahier de TD n° 3

THÈME 1 : MANIPULATIONS ET INITIALISATION DE POINTEURS

VRAI/FAUX

INSTRUCTIONS CORRECTES ET INCORRECTES : ADRESSES ET POINTEURS

PARTAGE DE CONTENU

VARIABLES LIÉES

QUE FAIT CE PROGRAMME ?

UN PROGRAMME UN PEU TORDU.

ARITHMÉTIQUE DES POINTEURS

THÈME 2 : ALLOCATION DYNAMIQUE, POINTEURS ET TABLEAUX

NOTATIONS * ET NOTATION []

INVERSION DE TABLEAU AVEC DES POINTEURS

ALLOCATION DYNAMIQUE DE TABLEAU À 1 DIMENSION

TABLEAUX DE CARACTÈRES ET ALLOCATION : COMMANDE DE DUPLICATION DE TEXTE

Thème 1 : manipulations et initialisation de pointeurs

Vrai/Faux

Répondez par vrai ou faux aux affirmations suivantes :

- Une variable peut ne pas avoir d'adresse **F toute variable a un emplacement mémoire (une adresse)**
- Soit x une variable d'un type simple (caractère, entier ou réel). La notation &x a un sens. **V**
- **Soit x une variable de type pointeur. La notation &x a un sens. V**
- Soit x une variable d'un type simple (caractère, entier ou réel). La notation *x a un sens. **F**
- **Soit x une variable de type pointeur. La notation *x a un sens. V**
- & signifie : adresse de **V**
- Soit x un pointeur. Alors, x est une variable. **F** (*car il existe des pointeurs constants, exemple les tableaux ...*)
- Soit p un pointeur. On peut toujours utiliser la notation *p dans un programme. **V**
mais attention PLANTAGE si pointeur non initialisé
- Un pointeur variable peut avoir comme valeur unique successivement plusieurs adresses. **V**
- Un pointeur variable peut avoir plusieurs emplacements mémoire, soit des adresses. **F**
- **long * p;** signifie : p est un pointeur de long, on dit que le pointeur est de **type long * V**
type variable
long * p1, * p2; // 2 pointeurs de longs
- *p signifie : contenu de p. Le contenu est de **type long. V**

Instructions correctes et incorrectes : adresses et pointeurs

Dans le programme suivant, indiquez quelles instructions sont incorrectes et pourquoi. Attention ! toutes les instructions sont correctes syntaxiquement (c'est à dire qu'elles sont bien écrites), donc un compilateur acceptera ce programme, mais le résultat aboutit souvent à un plantage du programme !

```
int main()
{
    long val1, val2; // valeurs aléatoires
    long * p_ent1, * p_ent2; // NON INITIALISES
                        ==> valeurs aléatoires
                        ==> p_ent1 et p_ent2 en ZONE INTERDITE
                        ==> plantage si accès au contenu des pointeurs
                        (valeurs déréférencées *p_ent1 et *p_ent2)
}
```

```
val1 = 5;
val2 = -125;



p_ent1 = p_ent2; // OK affectation autorisée mais ne sert à rien  

// toujours des valeurs aléatoires  

// p_ent1 et p_ent2 encore en ZONE INTERDITE



*p_ent1 = val2; // PLANTAGE ACCES EN ECRITURE *p_ent1 = ...



p_ent2 = &val1; // OK p_ent2 pointeur valide  

// pointe sur val1 et *p_ent2 vaut 5

*p_ent2 = val2; // OK *p_ent2 vaut -125  

// val1 a été modifiée et vaut -125

*p_ent1 = *p_ent2; // PLANTAGE *p_ent1 car p_ent1 en ZONE INTERDITE

p_ent1 = p_ent2; // OK p_ent2 pointe sur val1  

// donc p_ent1 pointe sur val1  

// donc *p-ent1 vaut -125



*p_ent1 = val1; // inutile déjà -125  

val2 = *p_ent2; // inutile déjà -125


}
```

Si l'on supprime du programme toutes les instructions incorrectes ou inutiles, quelles valeurs prendront les différentes variables ? (voir ci-dessus).

Partage de contenu

Variables liées

Ecrire un programme définissant une variable de type caractère, deux pointeurs vers des caractères, et qui fait en sorte que si l'on modifie le contenu de l'un des deux pointeurs ou la valeur de la variable, alors les contenus des deux pointeurs et de la variable prennent tous la même valeur, sans utiliser de test ni de boucle.

Faites une illustration des contenus et des valeurs des variables utilisées pour bien matérialiser l'effet des instructions utilisées.

Que fait ce programme ?

Indiquez les valeurs prises par les variables de ce programme :

```
int main()
{
double a,b,c;
double *p_a, *p_b;

a = 0.001;
b = 0.003;

p_a = &a;
*p_a = *p_a * 2.0;
p_b = &b;
c = 3.0 * (*p_b - *p_a);
}
```

un programme un peu tordu.

Notre ami Gilbert, toujours absent en amphithéâtre, a décidé d'écrire un programme, qui est correct, mais qui est vraiment tordu. Que fait ce programme ? Ecrivez-le de manière plus simple (en 4 à 5 lignes, normalement, c'est possible...)

```
int main()
{

    long a,b;
    long *ptr, *qtr;

    printf("entrez une valeur positive:");
    scanf("%ld",&a); // on suppose que la valeur entrée est
    positive
    b = 0;

    ptr = &a;
    qtr = ptr - (&a - &b);

    while (*qtr < *ptr)
    {
        *qtr = *qtr +1;
    }

    printf("et voilà, b =%ld\n",b);
}
```

Arithmétique des pointeurs

Rappelez les effets des instructions du programme suivant (on suppose que les variables concernées sont situées à des adresses consécutives en mémoire). Utilisez la représentation avec flèches au besoin. (un réel occupe 8 octets, un pointeur occupe 4 octets).

```
int main()
{
    double val_a, val_b, val_c;
    double *pdoub;
    double *qdoub;

    val_a = 0.0;
    val_b = 3.1415;
    val_c = 1.E-50;

    qdoub = &val_b; // types double *
    pdoub = qdoub; // types double *

    pdoub = pdoub-1; // types double *

    printf("%ld\n", qdoub-pdoub); // type long

    qdoub = qdoub+1; // types double *

    printf("%lf\n", *qdoub); // type double

    *qdoub = val_a; // types double

    *pdoub = (*pdoub)+1; // type double

    // type long * et type double
    print("%ld - %lf\n", qdoub-pdoub, *pdoub);

    *qdoub = *pdoub; // types double
    pdoub = pdoub+1; // types double *

    // type long et 2 types double
    printf("%ld %lf %lf\n", qdoub-pdoub, *pdoub, *qdoub);
}
```

Thème 2 : Allocation dynamique, pointeurs et tableaux

Notations * et notation []

Soit le programme suivant :

programme pointeurs_et_tableaux

rappel:

$\&a[i] \iff a + i \quad \text{long} *$
 $a[i] \iff *(a + i) \quad \text{long}$

```
long a[9] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
long *p; // à l'adresse 3036  $\iff$  &p
        // car a vaut 3000 et a et p contigus en mémoire
p = a; // &a[0]  $\iff$  a + 0  $\iff$  a ; p vaut 3000, *p vaut 12
```

Quelles valeurs ou adresses fournissent ces expressions ? On supposera que le tableau a est stocké à l'adresse 3000 en mémoire de l'ordinateur. Un long occupe 4 octets.

Google: table de précedence (priorité) des opérateurs en C support de cours CNRS

- a) $*p + 2 \iff (*p) + 2 \iff 12 + 2 = 14$
- b) $*(p+2) \iff p[2] \iff 34$
- c) $\&p+1 \iff (\&p) + 1$
 $3036 + 1 \text{ long} * \iff 3036 + 1 * \text{sizeof(long)} = 3040$
sizeof est built-in operator (évalué à la compilation)
- d)
 $\&a[4]-3 \iff 3016 - 3 * \text{sizeof(long)} = 3004$
 $\&a[4-3]$ soit $\&a[1]$
 $\&a[4]-3 \iff a + 4 - 3 \iff a + 1 \iff \&a[1]$
- e) $a+3 \iff 3012$ soit $\&a[0]+3$ soit $\&a[3]$
- f) $\&a[7]-p \iff (3028 - 3000) / \text{sizeof(long)} = 28/4 = 7$
 p vaut $\&a[0]$
 nombre de cases d'écart entre les indices 7 et 0
- g) $p + ((*p) - 10) \iff 3000 + (12 - 10) * \text{sizeof(long)} = 3008$
- h) $*(p + *(p+8) - a[7])$
 $*(p + 90 - 89)$
 $*(p + 1) \iff p[1] \iff a[1] \iff *(a + 1)$
 23

(énoncé tiré de : http://www.ltam.lu/Tutoriel_Ansi_C/prg-c97.htm#Heading206)

#include <iostream>

using namespace std;

int main()

{

long a[9] = {12, 23, 34, 45, 56, 67, 78, 89, 90};

long * p;

p = a;

long i;

cout << "Valeurs des éléments du tableau" << endl;

for (i = 0; i < 9; i++)

{

cout << a[i] << ' ';

}

cout << endl;

cout << "Adresses des éléments du tableau" << endl;

for (i = 0; i < 9; i++)

{

cout << &a[i] << ' ';

}

cout << endl;

cout << "*p + 2 vaut : " << *p + 2 << endl;

cout << "(p + 2)vaut : " << *(p + 2) << endl;

cout << "&p + 1 vaut : " << &p + 1 << endl;

cout << "&a[4] - 3 vaut : " << &a[4] - 3 << endl;

cout << "a + 3 vaut : " << a + 3 << endl;

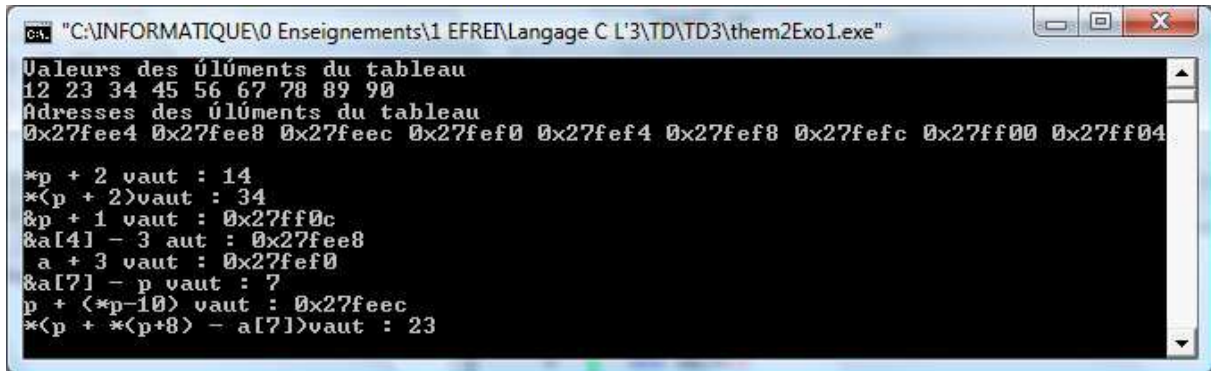
cout << "&a[7] - p vaut : " << &a[7] - p << endl;

cout << "p + (*p-10) vaut : " << p + (*p-10) << endl;

cout << "(p + *(p+8) - a[7])vaut : " << *(p + *(p+8) - a[7]) << endl;

return 0;

}



```
C:\INFORMATIQUE\0 Enseignements\1 EFREI\Langage C L'3\TD\TD3\them2Exo1.exe
Valeurs des éléments du tableau
12 23 34 45 56 67 78 89 90
Adresses des éléments du tableau
0x27fee4 0x27fee8 0x27feec 0x27fef0 0x27fef4 0x27fef8 0x27fefc 0x27ff00 0x27ff04
*p + 2 vaut : 14
*(p + 2)vaut : 34
&p + 1 vaut : 0x27ff0c
&a[4] - 3 aut : 0x27fee8
a + 3 vaut : 0x27fef0
&a[7] - p vaut : 7
p + (*p-10) vaut : 0x27feec
*(p + *(p+8) - a[7])vaut : 23
```

Inversion de tableau avec des pointeurs

Ecrire un programme qui range les éléments d'un tableau `tab` du type entier dans l'ordre inverse. Le programme utilisera des pointeurs `pDebut` et `pFin` et une variable `temp` pour la permutation des éléments.

```
#include <iostream>
#define TAILLE_UTILE 9

using namespace std;

int main()
{
    long a[TAILLE_UTILE] = {12, 23, 34, 45, 56, 67, 78, 89, 90};

    long i;
    cout << "Valeurs des éléments du tableau" << endl;
    for (i = 0; i < 9; i++)
    {
        cout << a[i] << ' ';
    }
    cout << endl;

    /** VERSION FACILE avec des indices i et j
        long j, tmp;

        for(i = 0, j = TAILLE_UTILE - 1; i < TAILLE_UTILE/2; i++, j--)
        // OU for(i = 0, j = TAILLE_UTILE - 1; i < j; i++, j--)
        {
            tmp = a[i];
            a[i] = a[j];
            a[j] = tmp;
        }
    */

    // VERSION avec des pointeurs pDebut et pFin
    long * pDebut, *pFin;
    long tmp;
    // Arithmétique des pointeurs transparente
    for( pDebut < a + TAILLE_UTILE/2; pDebut++, pFin--)
    // for(pDebut = a, pFin = a + TAILLE_UTILE - 1; pDebut < pFin; pDebut++, pFin--)
    {
        tmp      =      *pDebut;
        *pDebut  =      *pFin;
        *pFin    =      tmp;
    }

    cout << "Valeurs des éléments du tableau inversé" << endl;
    for (i = 0; i < 9; i++)
    {
        cout << a[i] << ' '; // *(a+i)
    }
    cout << endl;

    return 0;
}
```

Allocation dynamique de tableau à 1 dimension

Exo1: Ecrire un programme qui saisit une valeur entière n et alloue une zone de mémoire puisée dans le réservoir à octets(tas/heap) pour stocker n entiers, et l'affecte à un pointeur. Quelle est la taille maximum de ce tableau dynamique ? quelle est la taille utile de ce tableau dynamique ?

```
#include <iostream>

#define TAILLE_UTILE 9

using namespace std;

int main()
{
    long n;
    cout << "taille maximale du tableau : "; cin >> n;

    long * p;
    p = new long[n]; // requête
    if (p == NULL) // échec de la requête
    {
        cout << "Echec de l'allocation dynamique de mémoire" << endl;
        return 0;
    }

    cout << "La taille maximale du tableau est " << n << endl;
    cout << "La taille utile est inférieure ou égale à la taille maximale." << endl;

    return 0;
}
```

Exo2: Ecrire un programme qui effectue la saisie de n valeurs entières et les range dans la zone mémoire obtenue, que l'on peut considérer comme un tableau :

en utilisant la notation [] des tableaux

cin >> p[i];

en utilisant les notations avec des pointeurs

cin >> *(p+i)

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    long n;
```

```
    cout << "taille maximale du tableau : "; cin >> n;
```

```
    long * p;
```

```
    p = new long[n];
```

```
    if (p == NULL)
```

```
    {
```

```
        cout << "Echec de l'allocation dynamique de mémoire" << endl;
```

```
        return 0;
```

```
    }
```

```
    cout << "Saisie des " << n << " valeurs" << endl;
```

```
    for (long i = 0; i < n; i++)
```

```
    {
```

```
        cout << " valeur " << i << ' ';
```

```
        cin >> p[i]; // p[i] notation tableau ou *(p + i) notation pointeur
```

```
    }
```

```
    // parcours avec un pointeur courant (préparation des listes chaînées pas d'indice)
```

```
    for (long * pcourant = p; pcourant < p + n; pcourant++)
```

```
    {
```

```
        cout << *pcourant << ' ';
```

```
    }
```

```
    delete [] p;
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

Exo3: Ecrire un programme qui recherche une valeur dans ce tableau, en utilisant les notations des pointeurs. (pas de crochets [] donc !).

suite du code de l'exo2

```
long x;
cout << "entier recherché : ";
cin >> x;

long drapeau, * pcourant;
for (drapeau = 0, pcourant = p; (pcourant < p + n) && (drapeau == 0); pcourant++)
{
    if (*pcourant == x)
    {
        drapeau = 1;
    }
}

if (drapeau == 1)
{
    cout << x << " trouvé " << endl;
}
else
{
    cout << x << " non trouvé " << endl;
}
```

Exo4: On veut ajouter une nouvelle valeur à ce tableau. A quel problème se trouve-t-on confronté ?
Trouvez une méthode permettant, à partir du premier tableau (c'est à dire la zone mémoire obtenue par reservation()), de créer un tableau ayant une variable de plus dans laquelle on stockera la nouvelle valeur.

PB: tableau trop petit pour accueillir une nouvelle variable

Remède:

- 1) **allouer un nouveau tableau avec 1 élément de plus**
- 2) **recopier les valeurs de l'ancien tableau dans le nouveau**
- 3) **stocker la nouvelle valeur comme dernier élément du nouveau tableau**
- 4) **libérer la mémoire pour l'ancien tableau**

suite des exos 2 et 3

```
cout << "nouvelle valeur à ajouter : ";
cin >> x;
// allocation du nouveau tableau
long * p2;
p2 = new long[n+1];
if (p2 == NULL)
{
    cout << "Echec de l'allocation dynamique de mémoire" << endl;
    return 0;
}

// recopie des n valeurs
long i;
for (i = 0; i < n; i++)
{
    p2[i] = p[i]; // *(p2+i) = *(p+i)
}
p2[i] = x; // stockage de la nouvelle valeur à la dernière place

// Affichage avec un parcours avec un pointeur courant
for (pcourant = p2; pcourant < p2 + n + 1; pcourant++)
{
    cout << *pcourant << ' ';
}
cout << endl;

// libération du premier tableau
delete p;
```

Exo5: De la même manière, traitez le cas de la suppression d'une valeur d'un tableau, la valeur étant donnée par une saisie de l'utilisateur.

PB: tableau trop grand après la suppression d'une variable

Remède:

- 1) **rechercher la case à supprimer et la marquer une valeur "impossible" (pas besoin de décaler!)**
- 2) **allouer un nouveau tableau avec 1 élément de moins**
- 3) **recopier les valeurs de l'ancien tableau dans le nouveau en ignorant la case à supprimer**
- 4) **libérer la mémoire pour l'ancien tableau**

suite des exos 2, 3 et 4

```
// suppression d'une valeur dans le tableau p2
cout << "valeur à supprimer : ";
cin >> x;
for (i = 0; i < n + 1; i++)
{
    if (*(p2 + i) == x)
    {
        *(p2 + i) = LONG_MIN; // case marquée
        break;
    }
}

// allocation du nouveau tableau
long * p3;
p3 = new long[n];
if (p3 == NULL)
{
    cout << "Echec de l'allocation dynamique de mémoire" << endl;
    return 0;
}

// recopie des n valeurs
long j;
for (i = j = 0; j < n; i++, j++)
{
    if (p2[i] == LONG_MIN)
    {
        j--; // pour ne pas être en avance par rapport à i
        continue;
    }
    p3[j] = p2[i]; // *(p3+j) = *(p2+i)
}
```

```
}

// Affichage avec un parcours avec un pointeur courant
for (pcourant = p3; pcourant < p3 + n; pcourant++)
{
    cout << *pcourant << ' ';
}
cout << endl;

// libération du tableau
delete p2;
```


Tableaux de caractères et allocation : commande de duplication de texte

On cherche à programmer la commande de copie de texte d'un tableau vers un autre tableau en utilisant **d'abord des indices puis des pointeurs**.

On dispose d'un **tableau statique** de caractères (de taille maximum 100), et l'on souhaite dupliquer ce tableau dans un autre texte, dont **la longueur correspond exactement au nombre de caractères à stocker (donc tableau dynamique**, sans utiliser la commande strcpy.

```
#include <iostream>
#include <string.h>
#define TAILLEMAXIMALE 100
using namespace std;

int main()
{
    char source[TAILLEMAXIMALE] = "Copie d'un tableau de 100 caractères au plus";
    char * destination; // pointeur vers le tableau dynamique
    // pointeur à ne pas déplacer afin de pas perdre l'adress du début du tableau

    long tailleUtile = strlen(source) + 1; // 1 car '\0' en fin de tableau
    destination = new char[tailleUtile];
    if (destination == NULL)
    {
        cout << "problème d'allocation dynamique" << endl;
    }

    /**
    long i;
    for(i = 0; i < tailleUtile; i++) // ou source[i] != '\0' comme test d'arrêt
    {
        destination[i] = source[i]; // ou *(dest + i) = *(source + i)
    }

    // BUG car ça compile, on perd la destination, le pointeur n'est plus au début du tableau mais à la fin
    char * pi;
    for(pi = source; pi < source + tailleUtile; pi++, destination++)
    {
        *destination = *pi;
    }
    */

    // test d'arrêt
    // psource < source + tailleUtile (arithmétique des pointeurs)
    // OU *psource != '\0'
    char * psource, * pdestination; // 2 pointeurs variables
    for(psource = source, pdestination = destination; *psource != '\0'; psource++, pdestination++)
    {
        *pdestination = *psource;
    }
    cout << "copie du tableau : " << destination << endl;
    return 0;
}
```

RETOUR SUR LA FUSION TRIÉE DE DEUX TABLEAUX TRIES PAR ORDRE CROISSANT

Dans le TD2, le parcours des 3 tableaux était assuré par un jeu de 3 indices.

Dans le TD3, le parcours des 3 tableaux est assuré par un jeu de 3 pointeurs.

```
#include <iostream>

#define TAILLE_MAXIMALE1 5
#define TAILLE_MAXIMALE2 7

using namespace std;

int main()
{
    long tab1[TAILLE_MAXIMALE1];
    long tab2[TAILLE_MAXIMALE2];
    long tab3[TAILLE_MAXIMALE1 + TAILLE_MAXIMALE2];
    long tailleUtile1 = 0;
    long tailleUtile2 = 0;
    long tailleUtile3 = 0;
    long * ptr1 = NULL, * ptr2 = NULL, * ptr3 = NULL;

    cout << "Saisie des " << TAILLE_MAXIMALE1 << " valeurs du premier tableau par ordre croissant : " <<
endl;
    for (ptr1 = tab1; ptr1 < tab1 + TAILLE_MAXIMALE1; ptr1++)
    {
        cout << "entier : ";
        cin >> *ptr1;
        tailleUtile1++;
    }

    cout << "Saisie des " << TAILLE_MAXIMALE2 << " valeurs du second tableau par ordre croissant : " <<
endl;
    for (ptr2 = tab2; ptr2 < tab2 + TAILLE_MAXIMALE2; ptr2++)
    {
        cout << "entier : ";
        cin >> *ptr2;
        tailleUtile2++;
    }
}
```

```
cout << "Tableau1 : ";
for (ptr1 = tab1; ptr1 < tab1 + tailleUtile1; ptr1++)
    cout << *ptr1 << ' ';
cout << endl;

cout << "Tableau2 : ";
for (ptr2 = tab2; ptr2 < tab2 + tailleUtile2; ptr2++)
    cout << *ptr2 << ' ';
cout << endl;

// tant qu'il y a des éléments dans les 2 tableaux
for ( ptr1 = tab1, ptr2 = tab2, ptr3 = tab3;                // initialisation
      ((ptr1 < tab1 + tailleUtile1) && (ptr2 < tab2 + tailleUtile2)); // test
      ptr3++)                                              // action de fin de boucle
{
    if (*ptr1 <= *ptr2)
    {
        *ptr3 = *ptr1;
        ptr1++;
    }
    else // *ptr1 > *ptr2
    {
        *ptr3 = *ptr2;
        ptr2++;
    }
}

// tant qu'il y a encore des éléments dans le premier tableau
while (ptr1 < tab1 + tailleUtile1)
{
    *ptr3 = *ptr1;
    ptr1++;
    ptr3++;
}

// tant qu'il y a encore des éléments dans le second tableau
while (ptr2 < tab2 + tailleUtile2)
{
    *ptr3 = *ptr2;
    ptr2++;
    ptr3++;
}
```

```
}
```

tailleUtile3 = ptr3 – tab3; // arithmétique des pointeurs (nombre de cases sans se soucier de la taille en octets des cases

```
cout << "Tableau3 : " << endl;

for (ptr3 = tab3; ptr3 < tab3 + tailleUtile3; ptr3++)
    cout << *ptr3 << ' ';
cout << endl;
return 0;
}
```

REMARQUE:

Tout fonctionne parfaitement, mais il y a beaucoup de code redondant.

Le recours aux fonctions devient incontournable... (voir TD4).

ALLOCATION DYNAMIQUE 2D (cf TP3)

Ecrivez un programme qui réalise une allocation à deux dimensions pour un tableau 2D de dimensions respectives n et p (n lignes et p colonnes), où n et p seront saisis par vos soins. Il faudra bien vérifier que n et p sont strictement positifs.

La variable que vous utiliserez sera donc du type : `long **point_point;`

En utilisant les notations avec les crochets, remplissez le tableau ainsi obtenu alternativement avec des 0 et des 1, puis affichez son contenu.

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

int main()
{
    long ** matrice2D;
    long n = 5; // nombre de lignes
    long p = 6; // nombre de colonnes

    // allocation du tableau 1D de lignes, chaque ligne contient un pointeur de type long * vers un
    // tableau 1D d'entiers
    matrice2D = new long * [n];

    // allocations des n tableaux 1D de p entiers
    for (int i = 0; i < n; i++)
    {
        matrice2D[i] = new long [p];
    }

    // remplissage de la matrice alternativement avec des 0 et des 1
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < p; j++)
        {
            matrice2D[i][j] = j%2;
        }
    }
}
```

```
// affichage de la matrice
for (int i = 0; i < n; i++)
{
    for(int j = 0; j < p; j++)
    {
        cout << *( (*matrice2D + i) + j) << ' '; // <==> matrice2D[i][j]
    }
    cout << endl;
}
cout << endl;

return 0;
}
```

remarque:

posons **X** = matrice2D[i] = ***(matrice2D + i)**

alors on a:

$$\mathbf{matrice2D[i][j] = X[j] = *(X + j)}$$
$$= ***((*matrice2D + i) + j)** // A COMPRENDRE, NE PAS ECRIRE$$

CF:

void f(int tab[])

void f(int * tab)

void main(int argc, char arg [[]])

void main(int argc, char * arg [])

void main(int argc, char ** arg)