C++	Ch 7 – Pointeurs et structures dynamiques

I. POINTEURS **1**
- A. POINTEURS SUR TYPES DE DONNEES DE BASE 1
- B. POINTEURS SUR TYPES STRUCTURES 2
- C. POINTEURS SUR TABLEAUX 2
- D. POINTEURS DE FONCTIONS 3
- E. DOUBLE INDIRECTION 4
- F. DANGERS SUR L'UTILISATION DES POINTEURS 5

II. ALLOCATION DYNAMIQUE : NEW ET DELETE **5**
- A. SYNTAXE 5
- B. LISTES LINEAIRES (LISTES CHAINEES) 5
- C. TABLEAUX DYNAMIQUES 7

I. Pointeurs

A. Pointeurs sur types de données de base

A la déclaration d'une variable est associée la réservation d'un espace mémoire pouvant stocker le contenu de cette variable.
Un pointeur est une variable qui contient l'adresse mémoire d'une autre variable. On dit que le pointeur pointe sur la variable pointée, fait référence à la variable pointée (l'adresse mémoire où est stocké son contenu).
Les références (opérateur & déjà rencontré lors de la déclaration des passages de paramètres) introduites en C++ sont un moyen plus simple d'écrire les mécanismes utilisés par les pointeurs (introduits en C).

1. Déclaration d'un pointeur

L'opérateur * permet la déclaration d'un pointeur. La déclaration d'un pointeur précise toujours le type de la variable pointée.

```
int Ventier = 0 ; // déclaration d'une variable entière
int * ptrEnt; // déclaration d'un pointeur vers un entier
```

2. Initialisation d'un pointeur

L'opérateur **&** (**opérateur d'indirection**) permet la récupération de l'adresse d'une variable.

```
ptrEnt = &Ventier ; // ptrEnt pointe sur Ventier
```

3. Utilisation d'un pointeur

On peut utiliser un pointeur comme si on utilisait la variable pointée en utilisant l'opérateur * (**opérateur de déréférencement**) :

```
*ptrEnt=*ptrEnt + 1 ; // ajouter 1 au contenu pointé
```

B. Pointeurs sur types structures

On peut utiliser les pointeurs vers des données de type complexe.

1. Déclaration d'un pointeur

L'opérateur * permet la déclaration d'un pointeur. La déclaration d'un pointeur précise toujours le type de la variable pointée.

```
struct client
{
    int VnumCli ;
};
client VunClient ; // déclaration variable de type client
client *ptrVcli; // déclaration d'un pointeur vers client
```

2. Initialisation d'un pointeur

L'opérateur **&** (**opérateur d'indirection**) permet la récupération de l'adresse d'une variable.

```
ptrVcli = &VunClient ; // le pointeur pointe vers VunClient
```

3. Utilisation d'un pointeur

On peut utiliser un pointeur comme si on utilisait la variable pointée en utilisant l'opérateur ***** (**opérateur de déréférencement**) :

```
ptrVcli->VnumCli = 1 ;
```

ou bien :

```
(*ptrVcli).VnumCli = 1 ;
```

C. Pointeurs sur tableaux

On peut utiliser les pointeurs vers des données de type complexe.
L'adresse d'un tableau est l'adresse de son premier élément. L'utilisation d'un pointeur vers un tableau n'est donc utile que si on souhaite adresser plusieurs tableaux différents.

1. Déclaration d'un pointeur

L'opérateur * permet la déclaration d'un pointeur. La déclaration d'un pointeur précise toujours le type de la variable pointée.

```
int Tentier[100] ;
int *ptrTent; // déclaration d'un pointeur entier
```

2. Initialisation d'un pointeur

L'opérateur & (opérateur d'indirection) permet la récupération de l'adresse d'une variable.

```
ptrTent=Tentier; // le pointeur pointe vers Tentier
```

ou bien :

```
ptrTent=Tentier[0]; // pointe vers Tentier[0]
```

3. Utilisation d'un pointeur

On peut utiliser un pointeur comme si on utilisait la variable pointée en utilisant l'opérateur * (opérateur de déréférencement) :

```
ptrTent[4] = 10 ;
```

```
*(ptrTent+3) = 10 ;
```

est équivalent à :

```
Tentier[4] = 10 ;
*(Tentier+3) = 10 ;
```

D. Pointeurs de fonctions

On peut utiliser les pointeurs vers des données de type complexe.

1. Déclaration d’un pointeur

L’opérateur * permet la déclaration d’un pointeur. La déclaration d’un pointeur précise toujours le type de la variable pointée.

```
int fSomme(int Vnb1, int Vnb2)
{
    return (Vnb1+Vnb2) ;
};
int (*ptrf)(int, int); // decl. pointeur sur fonction
```

2. Initialisation d’un pointeur

L’opérateur & (opérateur d’indirection) permet la récupération de l’adresse d’une variable.

```
ptrf=&fSomme; // le pointeur pointe sur la fonction fSomme
```

3. Utilisation d’un pointeur

On peut utiliser un pointeur comme si on utilisait la variable pointée en utilisant l’opérateur * (opérateur de déréférencement) :

```
int V1 = 10;
int V2 = 20;
int V3 = (*ptrf)(V1, V2) ;
```

4. Un exemple intéressant

On peut utiliser un pointeur comme si on utilisait la variable pointée en utilisant l’opérateur * (opérateur de déréférencement) :

```
// définition de fonctions de calculs arithmétiques
int fSomme(int Vnb1, int Vnb2)
  {    return (Vnb1+Vnb2) ; }
int fDiff(int Vnb1, int Vnb2)
  {    return (Vnb1-Vnb2) ;}
int fProduit(int Vnb1, int Vnb2)
  {    return (Vnb1*Vnb2) ; }
int fQuotient(int Vnb1, int Vnb2)
  {    return (Vnb1/Vnb2) ;}
int fModulo(int Vnb1, int Vnb2)
  {    return (Vnb1%Vnb2) ;}

typedef int (*ptrf)(int, int); // décl. pointeur sur fonction

ptrf Tfonc[5] ; // allocation d’un tableau de pointeurs

int main (void)
```

```
{
    int V1, V2, V3 ;
    Tfonc[0]= &fSomme ;
    Tfonc[1]= &fDiff ;
    Tfonc[2]= &fProduit ;
    Tfonc[3]= &fQuotient ;
    Tfonc[4]= &fModulo ;

    cout << "entrez 2 nombres " ;
    cin >> V1 >> V2 ;

    cout << " entrez un numéro de fonction (0 à 4) " ;
    cin >> V3;

    cout << "  resultat = "  << (*(Tfonc[V3]))(V1,V2) );

    return 0 ;
}
```

E. Double indirection

On peut utiliser les pointeurs pour contenir l'adresses d'autres pointeurs

1. Déclaration d'un pointeur

Déclaration d'un pointeur vers un pointeur vers char :

```
char ** ptr ;
```

2. Initialisation d'un pointeur

Allocation dynamique et initialisation du pointeur :.

```
char ch1[] = "abc" ;
char ch2[] = "def" ;
ptr = malloc(sizeof(char) * 2);
ptr[0] = ch1;
ptr[1] = ch2;
```

3. Utilisation d'un pointeur

On peut utiliser un pointeur comme si on utilisait la variable pointée en utilisant l'opérateur * (opérateur de déréférencement) :

```
char c;
for(i=0;i<3;i++)
{
    c = *(*(ptr)+i); // contenu à l'adresse de ptr
    printf("%c",c);
}
printf("\n");
for(i=0;i<3;i++)
{
    c = *(*(ptr+1)+i); // contenu à l'adresse de ptr +1
    printf("%c",c);
}
```

F. DANGERS SUR L'UTILISATION DES POINTEURS

Les pointeurs permettent d'accéder à n'importe quel emplacement de la mémoire réservée à un programme : ils représentent un danger si il sont mal initialisés ou pas correctement utilisé.
Les pointeurs non initialisés (ou mal initialisés) sont la cause d'erreurs fréquentes d'exécution (erreur Windows).

II. Allocation dynamique : new et delete

A. Syntaxe

1. Allocation : new

Syntaxe :
IdentificateurType * identificateurPtr ; // déclaration
identificateurPtr = new IdentificateurType ; // allocation

```
int * ptr ;
ptr = new int() ;
*ptr=10 ;

char * mot ;
mot = new char[20] ;
```

2. Destruction : delete

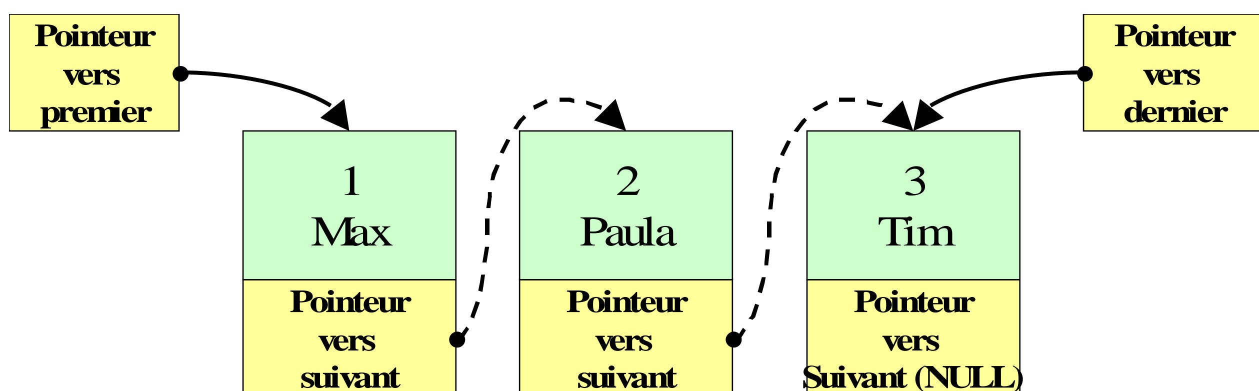
Syntaxe :
delete identificateurPtr ; // désallocation

```
delete ptr ;
delete [] mot ;
```

B. Listes linéaires (listes chaînées)

On connaît la limitation des tableaux déclarés avec un nombre d'indices déterminé.

Une liste chaînée est une structure de données constituée d'éléments chaînés, où chaque élément possède des **données propres** et un **pointeur vers l'élément suivant**. Le nombre d'élément de cette structure n'est pas limité (sauf par la taille²mémoire disponible)



1. Déclarer une structure de données

Déclarer la structure de données avec le pointeur sur l'élément suivant :

```
struct etudiant
   {
    int Vnum ;
    char Vnom[20] ;
    etudiant * ptrSuivant ;
   }
```

Déclarer les pointeurs qui vont permettre l'entrée dans la liste (ptrPrem) et d'accéder directement au dernier élément de la liste (ptrDern) :

```
etudiant * ptrPrem = NULL, * ptrDern = NULL;
```

Déclarer un pointeur de l'élément qu'on est en train de créer (ptrUnEtudiant) :

```
etudiant * ptrUnEtudiant = NULL;
```

2. Construire chaque élément et chaînage avec le suivant

Construire un nouvel élément « etudiant » et conserver un pointeur sur celui-ci ; gérer l'erreur d'allocation mémoire :

```
ptrUnEtudiant = new etudiant ;
if (ptrUnEtudiant==NULL)
   {
    cout << "erreur allocation mémoire " ;
    return(1) ;
   }
```

Initialiser les valeurs de l'élément :

```
ptrUnEtudiant ->Vnum = 1 ;
cout << "entrez un prenom : ";
cin >> ptrUnEtudiant ->VnomCli ;
ptrUnEtudiant ->ptrSuivant = NULL ;
```

Si c'est le premier élément, conserver un pointeur sur cet élément, sinon le dernier élément qui avait été créé pointe sur ce nouvel élément, et le dernier élément devient ce nouvel élément :

```
if (ptrPrem==NULL)
   {
    ptrPrem = ptrUnEtudiant;
    ptrDern = ptrUnEtudiant;
   }
else
   {
    ptrDern -> ptrSuivant = ptrUnEtudiant;
    ptrDern = ptrUnEtudiant;
   }
```

3. Parcourir la liste

Démarrer par le premier élément :

```
ptrUnEtudiant = ptrPrem;
```

Si le premier élément vaut NULL, la liste est vide, sinon parcourir la liste par le lien de chainage de pointeurs sur l'élément suivant ::

```
if (ptrUnEtudiant==NULL)
    cout << "liste vide";
else
    while(ptrUnEtudiant!= NULL)
        {
```

```
        cout << ptrUnEtudiant ->Vnom << endl;
        ptrUnEtudiant = ptrUnEtudiant ->ptrSuivant;
       }
```

C. Tableaux dynamiques

On connaît la limitation des tableaux déclarés avec un nombre d'indices déterminé.
Les pointeurs offrent une possibilité de créer dynamiquement des tableaux : le nombre d'éléments est donné par l'utilisateur (saisie d'un nombre au clavier).

1. Déclarer les variables et pointeur

Déclarer la structure de données avec le pointeur sur l'élément suivant :

```
int Vtaille, i ;
```

Déclarer le pointeur qui va devenir le tableau lui-même :

```
int * ptrTab ;
```

2. Construire l'espace réservé pour le tableau et pointer dessus

Initialiser les valeurs de l'élément, allouer l'espace :

```
cout << "entrez un nombre d'elements : ";
cin >> Vtaille ;
ptrTab = new int[Vtaille] ;
 if (ptrTab==NULL)
    {
     cout << "erreur allocation mémoire " ;
     return(1) ;
    }
```

3. Utiliser le tableau

Le tableau ainsi alloué peut être utilisé comme tout autre tableau :

```
for (i=0 ;i<Vtaille ;i++)
   {
    ptrTab[i]=i ;
    ...
   }
```

4. Libérer la mémoire

La destruction du tableau permet la libération de la mémoire qui lui avait été allouée :

```
delete [] ptrTab ;
```