Pointeurs – Adresse – Référence :

Un pointeur : Une variable prévue pour contenir une adresse

< Type pointeur > p ;

Type * p ;

Type2 * * p2 ;

Int * pi ;
int i ;
opérateur & (« Adresse de »)

pi = &i ; // pi contient l'adresse de i

Accès à l'objet pointé :

Opérateur *
* pi ⟸⟹ i

int T[10] ;
int *p1, *p2 ;
p1 = & T[2] ;
p2 = &T[5] ;

Remarque :

Type de T : CONST INT *
T contient l'adresse de la première case du tableau

→ Opérateur []
Exp1 [Exp2]

Pointeur Int

T[3]
p1[2] ⟸⟹ T[4]
* p1 ⟸⟹ T[2]

Remarque : []

Exp1 [Exp2]

T[3] ⟸⟹ 3 [T]

Opérations sur les pointeurs :

Addition d'un entier P1 + 3
Résultat : p1 + 3 = adresse 3 « int » plus loin

p1 + 3 : Adresse de T[5]

[] : T[3] ⇐⇒ * (T + 3)

Différence de pointeur
p1 – p2 → -3
p2 – p1 → +3

```
Struct Gens
{
        char nom[20]   ;
                int age ;
} S ;

gens * p ;
p = & S ;
```

Rappel :

→ ⇐⇒ (*) .

Référence :

« Un autre nom pour une variable »

Type &p ;

1. Utilisation bête :

```
int i ;
int & ri = i ;
i = 12 ;
cout<<ri ;      → 12
```

Contrainte :

Affectation seulement à l'initialisation

```
int i = 12, j =27 ;
int & ri = i ;
ri = 12 ;
ri = j ;
i = 27 ;
j = 39 ;
```

2. Utilisation intéressante de la déclaration par référence dans un passage d'argument à une fonction :

C++

```
void foixdeux( int &v) {       v = 2*v ;        }
int i = 12 ;
foisdeux( i ) ;
i = 24 ;
```

C

```
void foisdeuxC( int * v) {     v = (*v)*2 ;     }
foisdeuxC( &i ) ;
```

3. Retour de fonction par référence :

```
int & Mat( int i, int j ) ;

const int NL = 4 ;
const int NC = 10 ;
int M[ NL * NC ] ;
case 2,7 : M[ 2 * NC+ 7 ]
```

Première version :

```
int Mat( int i, int j )
{
        return M[i * NC + j ] ;
}

i = Mat( 2 , 7 ) ;          // OK
Mat( 2, 7 ) = 39            // Pas bon
```

Deuxième version :

```
int & Mat( int i, int j )
{
        return M[I * NC + j ] ;

}

i = Mat( 2, 7 ) ;           //OK
Mat( 2, 7 ) = 39 ;          //OK
```

Version :

```
int * MatC( int I, int j )
{
        &( M[I* NC + j ] ;
}

*( MatC( 2, 7) ) = 12 ;
```

Arbre binaire ordonné :

Insertion d'une valeur

Définition : C'est un arbre binaire où pour chaque nœud, toutes les valeurs de son sous-arbre gauche sont inférieures à la valeur du nœud et toutes les valeurs de son sous-arbres droit sont supérieures à la valeur du nœud.

```
void Arbre ::AjouterIt( int i )
{
        Nœud * n, *nn ;
        n = racine ;
        nn = NULL ;

        while( n != NULL )
        {
                nn = n ;
                if( i < n→info )
                        n= n → fg ;
                else
                {
                        if( i > n → info )
                                n = n→ fd ;
                        else
                                cout<<"Pas de double"<<endl ; return ;
                }
        }
        if( nn = NULL )
                racine = new Noeud( i, NULL, NULL ) ;
        else
                if( i < nn → info )
                        nn→fg = new Noeud( i, NULL, NULL ) ;
                else
                        nn→fd = new Noeud( i, NULL, NULL ) ;
}

void Arbre::AjouterIt2( int i )
{
        Noeud **n ;
        while( *n != NULL )
        {
                if( i < ( *n ) → info )
                        n = &( *n→fg ) ;
                else
                        if( i > ( *n ) → info )
                                n = &( *n→fd ) ;
                        else
                                cout<<"Pas de double"<<endl ; return ;
        }
        *n = new Nœud( i, NULL, NULL ) ;
}
```

Version avec référence :

```
void Arbre ::AjouterRef( int i )
{
        AjouterRef2( i, racine ) ;
}

void Arbre ::AjouterRef2( int i, Nœud * &n )
{
        if( n = NULL )
                n = new Noeud( i, NULL, NULL ) ;
        else
                if( i < n→ info )
                        AjouterRef2( i, n→fg ) ;
                else
                        if( i > n → info )
                                AjouterRef2( i, n→fd ) ;
                        else
                                cout<<"Pas de double"<<endl ; return ;
}



void Arbre ::AjouterAd( int i )
{
        AjouterAd2( i, &racine ) ;
}

void Arbre ::AjouterAd2( int i , Nœud ** n )
{
        if( * n = NULL )
                *n = new Noeud ( i, NULL, NULL ) ;

        if( i < ( *n ) → info )
                AjouterAd2( i, &((*n)->fg )) ;
        else
                if( i > (*n)→info )
                        AjouterAd2( i, &((*n)→fd ) ) ;
                else
                        cout<<"Pas de double"<<endl ; return ;
}
```

Entrées / Sorties / Fichiers :

1. **Fichier :** données plus ou moins organisées, plus ou moins lisibles(Visualisable avec un programme externe)

2. **Producteur et/ ou consommateur de données :**

iostream : - cin → bufferisé
- cout → bufferisé
- cerr → non bufferisé

#include <iostream.h>

fichier prédéfini cin, cout, cerr

Connexion d'un flux à un fichier
On a besoin de <fstream.h>

Type d'un flux : fstream

Connexion à un fichier :
Fonction membre : open

open(<FileName>, <Mode d'ouverture>)

<FileName> : char *
<Mode d'ouverture> : int

WINDOWS : "C:\\USERS\\TOTO\\…"
UNIX : "/ETD/IUP1/…"

Mode d'ouverture :

lire	ios ::in
Ecrire	ios ::out
Lire/ Ecrire	ios ::in |ios ::out

fstream f ;

f.open(« /ETD/IUP1/NAME/TOTO.DAT », ios ::in |ios ::out) ;

Ajouter	ios ::app	(Pour écrire à la fin du fichier)
Binaire	ios ::binary	(N'a pas d'effet sous WINDOWS ou UNIX)

Constructeur à deux arguments : → déclaration et « ouverture »

fstream f(« /ETD/… », ios ::in) ;

Déconnexion :

"Fermeture" fonction membre : close

f.close() ;

Sortie-Ecriture :

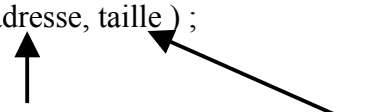
opérateur : << (Ecriture formatée)
cout<<'A' ;
f<<'A' ;

fonction membre put (écriture d'un caractère)
cout.put('A') ;

Fonction membre : write :

fstream f(« TOTO », ios ::out) ;

write(adresse, taille) ;



Adresse de ce que l'on veut écrire Nombre d'octet

Exemple : f.write(&i, sizeof(int)) ;

```
float    T[20] ;
f.write( T, sizeof(float) * 20 ) ;
char S[ ] = "Bonjour" ;
f.write( S, 8*sizeof( char ) ) ;
```

Entrées / Lecture :

Lecture formatée >>

flux>>référence ;

```
int i ;
cin>>i ;
```

« Espaces blancs » : Espace, tabulation, fin de ligne, retour chariot.

Non formatée : fonction read

```
read( Adresse, Taille ) ;
```

Adresse : adresse de l'objet où ranger les octets extrait du flux

Taille : Nombre d'octet à extraire

```
f.read( &i, sizeof( int) ) ;
```

Test sur un flot :

```
f.good() ;          ( 1 si c'est bon )
f.fail() ;          ( peut être qu'après cela peut s'arranger )
f.bad() ;           ( Irrécupérable )
f.eof() ;
```

Test du flot if(f) (1 si c'est bon)

Lecture (autre)

GetLine(char *S, int longueur, char *délimiteur*) ;
→ lit au maximum longueur –1 caractères puis les ranges à l'adresse donnée par S, et ajoute le caractère «\. » à la fin., on s'arrête si l'on rencontre le caractère *délimiteur* .

Vidange d'un flot de sortie : flush() ;