



**LA SURCHARGE DES OPÉRATEURS EN  
C++**

**LE TRAITEMENT DES EXCEPTIONS EN  
C++**

**LA CLASSE MATRICE TRIDIAGONALE**



# LA SURCHARGE DES OPÉRATEURS EN C++

# LE PRINCIPE DE LA SURCHARGE D'OPÉRATEUR

- En C++, tous les opérateurs peuvent être surchargés
- Toute nouvelle surcharge doit respecter l'arité de l'opérateur initial.
  - Exemple : toute surcharge de l'opérateur + doit être un opérateur binaire.
  - Exemple : toute surcharge de l'opérateur ! (opérateur de la négation) doit être un opérateur unaire.
- L'appel d'un opérateur surchargé se fait comme l'opérateur initial.
  - Pour un opérateur binaire, les opérandes sont de part et d'autre de l'opérateur.
  - Pour un opérateur unaire, l'opérande est après l'opérateur.
  - Cas particulier : [ ] est opérateur binaire. Le premier opérande est à gauche du symbole [ et le deuxième opérande est entre les deux symboles [ ] .



# LA SYNTAXE POUR DÉFINIR UNE SURCHARGE

- Une surcharge d'opérateur en tant que fonction indépendante.
  - `TypeRetour operator nomOperateur (TypeOperande ope1,...) { ... }`
  - Exemple de définition : On suppose avoir défini la classe `CoupleReel` formée de deux réels (`double`)  
`CoupleReel operator + (CoupleReel c1, CoupleReel c2) { ... }`  
`CoupleReel operator * (double l, CoupleReel c) { ... }`
  - Exemple d'appels :  
`CoupleReel c1(2.0,3.5), c2(1.25, 4.3), c3, c4;`  
`double lambda = 4.2;`  
  
`c3 = c1 + c2;`  
`c4 = 4.2 * c1;`



# LA SYNTAXE POUR DÉFINIR UNE SURCHARGE

- Une surcharge d'opérateur binaire en tant que méthode d'une classe
  - Pour un opérateur binaire, la méthode n'a qu'un seul paramètre : le deuxième. L'objet de la classe qui appellera l'opérateur sera le premier opérande.
  - Exemple de définition : On suppose avoir défini la classe CoupleReel formée de deux réels (double)

```
class CoupleReel {  
    CoupleReel operator + (CoupleReel c2) { ... }  
}
```

```
CoupleReel c1(2.0,3.5), c2(1.25, 4.3), c3, c4;
```

```
c3 = c1 + c2; // équivalent à c3 = c1.operator+(c2);
```

- La surcharge de l'opérateur \* du slide précédent ne peut se faire que comme fonction indépendante car le premier opérande n'est pas un CoupleReel mais un double.



## LA SURCHARGE DE L'OPERATEUR =

- En C (donc en C++), l'opérateur = peut être appelé en cascade : `a = b = c = ...; .`
- Donc l'opérateur = est non seulement un opérateur binaire mais en plus il renvoie une information.
- Dans le cas d'une cascade, l'exécution se fait de droite à gauche.
- Dans notre exemple, l'exécution de `b = c` a donné lieu à deux actions
  - l'affectation du contenu de c dans b
  - le retour de c pour l'exécution de la prochaine affectation



# LA SURCHARGE DE L'OPERATEUR =

- En résumé le prototypage de la surcharge de = est :

Type & operator = (Type & op1, Type & op2) // cas indépendant

Type & operator = (Type & op2) // cas méthode d'une classe

Le symbole & signifie que le passage de paramètres et le retour se font par référence et non par copie comme habituellement en C.

- Code type de la surcharge de l'opérateur = dans une classe

```
Type & operator = (Type & op2) {
```

```
    // on affecte dans l'objet pointé par this le contenu de op2
```

```
    ...
```

```
    return op2;
```

```
}
```





# LA CLASSE MATRICE TRIDIAGONALE



# MODÉLISATION

- Une matrice tridiagonale est définie par :
  - La dimension : **dim**, un entier
  - Ses trois diagonales:
    - **diagCentral** : la diagonale principale
    - **diagL**: la diagonale sous la principale
    - **diagU**: la diagonale au-dessus la principale



# MODÉLISATION

- Chaque diagonale est un tableau de flottants dont la dimension dépend de celle de la matrice:
  - `diagCentral` : dim
  - `diagL`: dim-1
  - `diagU`: dim-1
- Pour faciliter la gestion dynamique de mémoire on utilisera la classe standard `<valarray>` pour les représenter
  - `valarray<double> diagCentral` : la diagonale principale
  - `valarray<double> diagL`: la diagonale sous la principale
  - `valarray<double> diagU`: la diagonale au-dessus la principale



# PREMIÈRE DÉCLARATION

```
class MatriceTriDiagonale
{
public:

    MatriceTriDiagonale();
    virtual ~MatriceTriDiagonale();
    // constructeur et destructeur par défaut

private:

    int dim;
    valarray<double> diagCentral;
    valarray<double> diagL;
    valarray<double> diagU;
};
```



# UN PREMIER CONSTRUCTEUR

```
class MatriceTriDiagonale  
{  
public:
```

```
.....
```

```
//constructeur à partir de la données des diagonales
```

```
MatriceTriDiagonale(int ,valarray<double>, valarray<double>,  
    valarray<double>);
```

```
//les accesseurs de base
```

```
valarray<double> getDiagCentral();
```

```
valarray<double> getDiagL();
```

```
valarray<double> getDiagU();
```

```
int getDim();
```

```
private:
```

```
.....
```

```
};
```



## EXERCICE: COMPLÉTER CE PREMIER CONSTRUCTEUR

MatriceTriDiagonale

```
(int n, valarray<double> C, valarray<double> L, valarray<double> U)
{
    dim=n;
    if( // vérification de la conformité des dimensions )
    {
        cout<<"erreur: les dimnsions sont incopatibles!\n";
    }
    else
    {
        /* Important! Il faut d'abord réserver la mémoire nécessaire
        *          avant d'affecter par copie un vecteur
        * On utilise ici la fonction resize() de la classe valarray
        */
        /*
        *Ensuite on affecte les valeurs aux diagonales
        */
    } }
}
```



# POUR ACCÉDER AUX MEMBRES PRIVÉS

```
valarray<double> MatriceTriDiagonale::getDiagCentral()  
{  
    return diagCentral;  
}
```

```
valarray<double> MatriceTriDiagonale::getDiagL()  
{  
    return diagL;  
}
```

```
valarray<double> MatriceTriDiagonale::getDiagU()  
{  
    return diagU;  
}
```

```
int MatriceTriDiagonale::getDim()  
{  
    return dim;  
}
```



# OPÉRATIONS SUR LES MATRICES

- Si A , B et C sont trois matrices de classe **MatriceTriDiagonale** comment programmer l'opération

$$C=A+B?$$

- Première solution: réaliser une fonction membre  
`MatriceTriDiagonale Somme(MatriceTriDiagonale)`

- On pourra alors écrire

$$C=A.Somme(B);$$



# OPÉRATIONS SUR LES MATRICES

- Autre solution: surdéfinir l'opérateur « + »:

```
MatriceTriDiagonale operator +(MatriceTriDiagonale A)
```

```
{  
    /* Création des trois diagonales de la matrice somme /
```

```
    valarray<double> C(dim);
```

```
    valarray<double> L(dim-1);
```

```
    valarray<double> U(dim-1);
```

```
    /* On utilise ici l'opérateur "+" qui est déjà surdéfini pour la classe valarray  
    * pour additionner les diaginles correspondantes */
```

```
    C=diagCentral+A.getDiagCentral();
```

```
    L=diagL+A.getDiagL();
```

```
    U=diagU+A.getDiagU();
```

```
    // on crée la nouvelle matrice et on la revoie
```

```
    MatriceTriDiagonale M(dim,C,L,U);
```

```
    return M;
```

```
}
```





# POUR TESTER

```
int main( int argc, char** argv)
{
    int dim=4;
```

```
    valarray<double> C( 1,dim); //Crée un vecteur et pose chaque
                                //      élément égal à 1
```

```
    valarray<double> A( 2,dim-1);
```

```
    valarray<double> B( 3,dim-1);
```

```
    MatriceTriDiagonale M(dim, C,A,B);
```

```
    MatriceTriDiagonale N(dim, C,B,A);
```

```
    MatriceTriDiagonale H=M+N;
```

```
    return 0;
}
```



# EXERCICE

- Surdéfinir l'opérateur « \* » pour la multiplication par un vecteur:

```
valarray<double> operator *(valarray<double> y)
{
    .....
}
```



# ACCÈS AUX ÉLÉMENTS D'UNE MATRICE

- Si A est une matrice de classe `MatriceTriDiagonale` peut on utiliser la syntaxe

$x=A[i][j]$ ?

- Ou

$A[i][j]=x$ ?

- Nous n'avons stocké que les éléments des trois diagonales principales qui se caractérisent par la condition suivante:

$$|i-j| \leq 1$$

- Que doit il se passer si l'on essaie d'accéder aux autres éléments?



# ACCÈS AUX ÉLÉMENTS D'UNE MATRICE

- si  $|i-j| > 1$  lors d'un accès « en lecture » de type
$$x=A[i][j]$$
- On devrait pouvoir obtenir 0 car c'est bien la valeur « implicite » d'un tel élément
- Par contre, il est impossible d'écrire une valeur à cet emplacement car il n'est pas prévu dans la modélisation de la classe



# ACCÈS AUX ÉLÉMENTS D'UNE MATRICE

- Nous allons réaliser deux opérateurs d'accès aux éléments
- Le premier permettra d'utiliser, uniquement en lecture, la syntaxe

$$x=A[i][j]$$

- Le second, avec une notation différente, permettra d'accéder, en lecture et en écriture, aux éléments des trois diagonales principales; il sera donc possible d'écrire

$$A(i,j)=x$$

A condition que  $|i-j| \leq 1$



# GESTION DES EXCEPTIONS

- Nous allons commencer par préparer la gestion des cas où l'accès à un élément de la matrice est impossible:
  - Soit parce que l'un des deux indices (i,j) dépasse les limites [0,dim-1]
  - Soit parce que nous tentons d'accéder en écriture à un élément tel que

$$|i-j| > 1$$



# GESTION DES EXCEPTIONS

- Il est possible d'utiliser la classe **exception** standard et les quelques classes dérivées
- Nous pouvons également créer une simple classe dérivée de exception pour personnaliser la gestion des différents cas qui peuvent se produire dans notre classe



# UNE CLASSE POUR GÉRER LES EXCEPTIONS

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

```
class Except_matrice : public exception
{
public:
    Except_matrice( char * message);
    const char* what();
private:
    char * message_err;
};

    Except_matrice(char * message) {
// TODO Auto-generated constructor stub
    message_err=message;
}

    const char  what()
    {
        return message_err;
    }
```






## ACCÈS AUX ÉLÉMENTS: OPÉRATEUR []

- L'opérateur [] ne prend qu'un seul argument
- Nous allons donc le définir de façon à ce qu'il revoie un tableau de valeurs (**valarray**) correspondant à une ligne entière de la matrice
- Les éléments de la ligne qui ne sont pas sur l'une des trois diagonales seront nuls



# ACCÈS AUX ÉLÉMENTS: OPÉRATEUR []

```
valarray<double> operator [] (int i)
{
    //contrôle des dimensions
    if(i<0 | i>=dim)
    {
        //On crée un objet exception pour récupérer le message sur la nature du
        // problème
        Except_matrice err("indice hors dimension de la matrice");
        //on déclenche une exception à l'aide de l'opérateur throw
        throw err;
    }
    else
    {
        .....
    }
}
```



# EXERCICE: COMPLÉTER LE CODE

```
valarray<double> operator [] (int i)
{ //contrôle des dimensions
  if(i<0 | i>=dim)
  { //On crée un objet exception pour récupérer le message sur la
    nature du problème
    Except_matrice err("indice hors dimension de la
      matrice");
    //on déclenche une exception à l'aide de l'opérateur throw
    throw err;
  }
  else{
    valarray<double> ligne(dim);
    .....
    return ligne;
  }
}
```



# UTILISATION PRATIQUE DE L'OPÉRATEUR []

- Une méthode pour afficher la matrice

```
void MatriceTriDiagonale::afficher(){  
    /* Affiche sur console la matrice ligne par ligne . A utiliser avec des  
       petites matrices! */  
    for(int i=0;i<dim;i++)  
    {  
        for(int j=0;j<dim;j++)  
        {  
            /* On utilise ici la surdéfinition de l'opérateur [] */  
            cout<<" " <<(*this)[i][j]<<" ";  
            // attention! En C++ this est un pointeur!  
        }  
        cout<<"\n";  
    }  
    cout<<"\n";  
}
```

- **Vous pouvez utiliser cette méthode pour afficher la somme de vos deux matrices**



# ACCÈS AUX ÉLÉMENTS: OPÉRATEUR ()

```
double & operator()(int i, int j)
{ //contrôle des dimensions
  if(i<0 | i>=dim | j<0 | j>dim-1)
  {
    //On crée un objet exception pour récupérer le message sur la nature du problème
    Except_matrice err("indice hors dimension de la matrice");
    //on déclenche une exception à l'aide de l'opérateur throw
    throw err;
  }
  else
  {
    if(abs(i-j)>1)
    {
      Except_matrice err("accès non autorisé aux éléments non diagonaux");
      throw err;
    }
    else{.....}
  }
}
```



# ACCÈS AUX ÉLÉMENTS: OPÉRATEUR ()

```
double> operator()(int i, int j)
{ //gestion des exceptions
  if(...)
  { ... }
  else{
    if(...)
    { ..... }
    else{ // quand tout va bien
      if(i==j) return diagCentral[i]; //sur la diagonale
      if(i>j)  return diagL[j]; // sous la diagonale
      else    return diagU[i]; // au-dessus de la diagonale
    }
  }
}
```



# ACCÈS AUX ÉLÉMENTS: OPÉRATEUR ()

- Pour tester, complétez votre main():

```
try{  
H(1,1)=10.1;  
H(2,3)=-5.3;  
}  
catch(Except_matrice &err)  
{  
cout<< err.what()<<"\n";  
}  
H.afficher();
```



# ALGORITHME DE THOMAS: EXERCICE

```
void Thomas(valarray<double> y, valarray<double> *x)
{
  Compléter ici
}
```





# ALGORITHME DE THOMAS: POUR TESTER

- Dans votre main():
  - Créez une matrice tri diagonale  $M$  de dimension 5 dont la diagonale centrale est constante, égale à 1, la diagonale gauche est égale à 2 et la diagonale droite à 3
  - Créez un vecteur  $Y$  de dimension 5, toutes composantes égales à 2
  - Trouvez  $X$  tel que  $MX=Y$  en utilisant la méthode Thomas
  - Vérifiez le résultat en utilisant l'opérateur de multiplication par un vecteur: comparez  $M*X$  et  $Y$

