Cours 3 - C++ pour les mathématiques appliquées

Opérateurs

La dernière fois ... -

Généralités sur le `C++`

... aujourd'hui

- Constructeur(s)
- Opérateur(s)

Constructeur/Destructeur

Constructeur

- Il est fortement conseillé d'initialiser un objet à l'aide d'un constructeur `class`(arguments)
- Un constructeur est généralement déclaré `public`

Constructeur : exemple

```
class Polynome {
  public :
    int degree;
    Polynome(int d); // declaration du constructeur
  private :
    double *coeffs_;
};

Polynome::Polynome(int d) {       // implementation externe
  degre = d;
  coeffs_ = new double[degre];    // allocation
  for(int i=0;i<degre;i++) coeffs_[i] = 0;
}
int main() {
  Polynome p(3);              // creation d'un polynome
  Polynome m=Polynome(3);  // creation d'un second polynome
}
```

Surcharge de constructeur

```
struct Polynome {
  double *coeffs_;                  // public par defaut
  unsigned int degree;
  Polynome(int d, double v=0);      //constructeur (degree, val)
  Polynome(const Polynome & P);     // constructeur par copie
};
Polynome::Polynome(unsigned int d, double val) {
  degree=d ;
  if (degree == 0) return ;
  coeffs_=new double[degree];
  for(int i=0;i<degree; i++) coeffs_[i]=val;
}
Polynome::Polynome(const Polynome & P) {                  // par copie
  degree=P.degree;
  if (degree == 0) return ;
  coeffs_=new double[degree];
  for(int i=0;i<degree;i++) coeffs_[i]=P.coeffs_[i]; //recopie
}
```

Constructeur par copie

- Par défaut, un constructeur par recopie est toujours créé. Mais ce constructeur se contente de recopier les membres bit à bit.
- Il ne se préoccupe pas des zones mémoires allouées par l'utilisateur.
- Le constructeur par recopie est invoqué implicitement lors du transfert d'objet comme argument d'entrée ou de retour.
- `Polynome P=R` appelle le constructeur par copie.

Constructeur : utilisation

```cpp
int main() {
  Polynome p1;                   // creation d'un Polynome par défaut
  Polynome p2=Polynome(3,0);     // creation d'un Polynome par valeurs
  Polynome p3=p1;                // creation d'un Polynome par recopie
  Polynome *pp = new Polynome(3); // pointeur sur un Polynome
}
```

Appel de constructeur

- Lorsqu'un objet est initialisé, un constructeur approprié est appelé.
- Les arguments de la liste d'initialisation sont fournis au constructeur.
- Syntaxe : `class_type identifiant(arguments`; ou `class_type identifiant{arguments`;`
- Les différentes formes d'initialisation
  - défaut : `Polynome f;`
  - avec valeur : `Polynome {}`;
  - directe : `Polynome f(42);`
  - liste : `Polynome f{42}`;`
  - copie : `Polynome f=g;`

Destructeur

- Lorsque un objet est détruit, il libère l'espace qu'il a crée pour stocker des membres mais pas l'espace alloué par l'utilisateur. On peut à l'aide d'un destructeur libérer cette espace mémoire.

```
class Polynome {
  double *coeffs_;
public:
  int degree;

  Polynome(int d);
  ~Polynome();
}

Polynome::~Polynome() {
  delete [] coeffs_;
}
```

- Un destructeur n'a jamais d'argument et est public.

Surcharge d'opérateur

Syntaxe de la surcharge

- On utilise le mot réservé `operator` et la syntaxe suivante

```
argRetour nomOperateur (argtEntree)
```

- Le nombre d'arguments d'entrée est lié au type d'opérateur (unaire, binaire, ternaire)
- L'argument de sortie dépend des objectifs fixés.

Exemple de surcharge

```
class Polynome {
  public :
    int degree;
    double *coeffs_;
    Polynome(int d, double v=0);    //constructeur (degreension, val)
    Polynome(const Polynome &P);       //constructeur par copie
    Polynome operator * (double d); //produit par un double
};

Polynome Polynome::operator *(double d) {
  Polynome Q(degree);
  for(int i=0;i<degree;i++) {
    Q.coeffs_[i] = coeffs_[i]*d;
  }
  return Q;
}
```

Exemple d'opérateur `=`

- Etape 0 : L'opérateur d'affectation doit faire correspondre une variable membre de la classe cible à une variable membre de la classe source.

```
void Polynome::operator=(Polynome P) {
  degree = P.degree;
  coeffs_ = new double[degree];
  memcpy(coeffs_, P.coeffs_, degree * sizeof(double));
}
```

- Etape 1 : L'affectation doit seulement être utilisé pour modifier la cible. La source doit rester inchangée :

```
void operator=(const Polynome P);
```

- Etape 2 : Afin d'améliorer la vitesse d'accès aux données de la variable source, l'argument peut-être passé par référence.

```
void operator=(const Polynome &P);
```

Exemple d'opérateur = (2)

- Etape 3 : Lors d'une affectation, la valeur de la variable source est donnée à la variable gauche. Ainsi, après l'opération, le type de retour doit être non- `void` .
  L'opération doit donc retourner une référence sur le type de l'objet.

```
Polynome &operator=(const Polynome &P);
```

- Etape 4 : Puisque le type de retour n'est plus `void` , l'opérateur doit retourner une valeur. L'opérateur d'affectation est fait entre deux objets de même type. L'opérateur doit donc retourner la même variable que celle qui l'a appelé.

Exemple d'opérateur : $=$

On utilise donc l'opérateur `this`. L'implémentation finale est donc

```
Polynome& Polynome::operator=(const Polynome &P) {
  degree = P.degree;
  coeffs_ = new double[degree];
  memcpy(coeffs_, P.coeffs_, degree * sizeof(double));

  return *this;
}
```

Il est important, dans certaines configuration de

- Vérifier la mémoire : est-ce que le tableau `coeffs_` est déjà alloué?
- Vérifier l'auto-référence.

Surcharge externe

- Comment fonctionne l'opération (double x Polynome)

```
3*P (double x  Polynome)
```

- Surcharge externe

```
Polynome operator *(double d, const Polynome & P) {
  Polynome Q(degree);
  for (int i=0;i<degree;i++) {
    Q.coeffs_[i] = P.coeffs_[i]*d;
  }
  return Q;
}

int main() {
  Polynome M(3);
  Polynome P=M*2; // surcharge de * interne
  Polynome Q=3*M; // surcharge de * externe
}
```

Surcharge externe (2)

- Pas d'ambiguïté : arguments d'entrée distincts entre les deux versions.
- Dans l'écriture `M*2` ou `M*3` , 2 et 3 sont des entiers: il y a conversion automatique en double.

Surcharge interne ou externe

- Version externe : `Polynome * double`

```
Polynome operator *(double d, const Polynome & P) {
  Polynome Q(degree);
  for (int i=0;i<degree;i++) {
    Q.coeffs_[i] = P.coeffs_[i]*d;
  }
  return Q;
}
```

- Version externe : `double * Polynome`

```
Polynome operator *(const Polynome & P,double d) {
  Polynome Q(degree);
  for (int i=0;i<degree;i++) {
    Q.coeffs_[i] = P.coeffs_[i]*d;
  }
  return Q;
}
```

Choisir le type de surcharge

- Mettre en externe si l'opération n'induit pas de modification de l'objet.
- Meilleure visibilité et cohérence.
- Ne fonctionne pas si `coeffs_` est protégé (`private`)

Surcharge fortement recommandée

- La surcharge de `*=` doit être interne.

```
class Polynome {
public:
  ...
  Polynome &operator*=(double d); //produit par un double
};
Polynome &Polynome::operator*=(double d) {
  for (int i = 0; i < degree; i++) {
    coeffs_[i] *= d;
  }
  return *this;
}
```

- On retourne une référence à l'objet lui même (on évite la recopie).

Liste des opérateurs surchargeables

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td><td>~</td><td>|</td></tr>
<tr><td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>++</td><td></td></tr>
<tr><td>!</td><td>=</td><td>==</td><td>!=</td><td><</td><td>></td><td><=</td><td>>=</td><td>=|</td></tr>
<tr><td>>></td><td><<</td><td><<=</td><td>>>=</td><td>&&</td><td>||</td><td>[]</td><td>()</td><td>,</td></tr>
<tr><td>new</td><td>new[]</td><td>delete</td><td>delete[]</td><td>--</td><td>->*</td><td></td><td></td><td></td></tr>
</table>

Opérateurs spéciaux

- Opérateurs qui ne peuvent être surchargés :
  - `::` résolution de portée
  - `.` sélection de membre
  - `.*` sélection de membre via un pointeur de fonction
  - `:?` opérateur ternaire
- Surcharges de `new` et `delete` pour réécrire les mécanismes d'allocation dynamique.

Surcharge rationnelle

- Utiliser le sens usuel des opérateurs et éviter des surcharges fantaisistes, par exemple la division de 2 vecteurs.
- Éviter toute confusion possible, par exemple `()` et `[]` sont des opérateurs d'accès pour des objets de type vecteur
    - `()` suivant la convention, commence à 0 ou à 1 : logique `C++` ou mathématique
    - `[]` commence à 0 : logique du `C++`

Exemple de surcharge simple

```
class vect {
  public :
    double *val; //tableau de valeurs
    ...
    double & operator()(int i){return val[i+1]};
    double & operator[](int i){return val[i]};
}
```

- Retourne une référence $\Longrightarrow$ modification d'un élément.

Opérateur de conversion

Cas particulier de l opérateur de conversion/transtypage.

```
operator Polynome()(const vect &);
```

Syntaxe générale

- Pas d'arguments de retour, s'apparente à un constructeur
- Est utilisé implicitement par le compilateur si nécessaire : Fonction attendant un Polynome et recevant un vecteur.
- Utilisation très délicate, augmente les ambiguïtés, à utiliser avec parcimonie.
- Il est plus simple et fortement recommandé d'utiliser un constructeur de la classe `Polynome`

```
Polynome(const vect &);
```

Opérateurs de flux

- Généralement les opérateurs `<<` et `>>` sont réservés à des opérations d'entrée/sortie dans un flux (écran, fichier, buffer, . . .).
- Flux standards
  - `cin` entrée clavier, classe `istream`
  - `cout` sortie écran, classe `ostream`
- Pour afficher un Polynome :
  `cout<<p` $\Longrightarrow$ `Polynome: x^2+x+1`

Opérateur de flux : exemple

```
class Polynome {
  ...
};
ostream &operator<<(ostream &out, const Polynome &P) {
  out << "Polynome : ";
  for (int i = 0; i < P.degree; i++) {
    out << P(i) << " ";
  }
  out << endl;
  return out;
}
istream &operator>>(istream &in, const Polynome &P) {
  for (int i = 0; i < P.degree; i++) {
    in >> P(i);
  }
  return in;
}
```

Flux: insertion dans une liste

- Insertion d'un Polynome dans une liste de polynomes

```
class liste_Polynome {
  public:
  Polynome ** liste ;                 // tableau de pointeurs
  unsigned int nb_Polynomes ;         // nombre de polynomes
  liste_Polynome(unsigned int i=10); // constructeur
  ~liste_Polynome ( ) ;               // destructeur
  void allonge ();                    // augmente la taille du tableau de 10
  Polynome & operator(const int &);  // acces au i eme polynome
  add(const Polynome &);              // ajout d'un polynome
};

liste_Polynome & operator <<(liste_Polynome &L,const Polynome &P) {
  L.add(P);
  return L;
}
```

Appel

Dans le programme principal

```
Polynome P();
liste_Polynome liste();
liste<<P;
```

Exemple complet de surcharge : définition

```
class Polynome
{
  public :
    ...
    double& operator()( int i);
    Polynome & operator += (const Polynome &);
    Polynome & operator -= (const Polynome &);
    Polynome & operator *= (const Polynome &);
    Polynome & operator /= (const Polynome &);
    bool operator == (const Polynome &);
    bool operator != (const Polynome &);
};

Polynome operator+(const Polynome &,const Polynome &);
Polynome operator-(const Polynome &,const Polynome &);
ostream & operator <<(ostream &,const Polynome &);
istream & operator >>(istream &,const Polynome &);
```

Exemple complet de surcharge : implémentation

```
Polynome & Polynome::operator+= (const Polynome &Q) {
  if(dim!=Q.degree) { exit(-1); // dimension incompatible
  }
  Polynome &P=*this; // alias sur this
  for(int i=0;i<degree;i++)
  { P(i)+=Q(i); }
  return P;
}

Polynome operator+(const Polynome P &,const Polynome &Q) {
  Polynome R(P); // R initialise avec P
  R+=Q; // addition interne
  return R; // retourne l'objet R
}

bool Polynome::operator == (const Polynome &`){...}
bool Polynome::operator != (const Polynome &Q){return ! (*this == Q);}
```

Exemple complet de surcharge : commentaires

- l'opération `P+M` appelle `+=`
- une seule implémentation de `+`.
- plus facile à maintenir.
- plus robuste : diminue le nombre d'erreur possible.

Un exemple plus évolué : définition

- Supposons que l'on dispose d'une classe de matrices `(MATRICES)` et que l'on veuille faire une combinaison linéaire de plusieurs matrices de grande taille.
- Problème : le calcul de `2*A+4*D-5*C` :
  `T1=5*C` , `T2=4*D` , `T3=T1-T2` , `T4=2*A` et enfin `T4+T3`

  soit 5 générations de matrices temporaires avec recopie !

Un exemple plus évolué : définition (2)

- Solution : utiliser une structure intermédiaire (combinaison) décrivant la combinaison linéaire et n'effectuer le calcul qu'au moment du `=` :

  `5*C=>C1` , `4*D=>C2` , `C3=C1-C2` , `2*A=>C4` et enfin `C4+C3 =>` matrice résultat.

- Les opérations créent ou modifient la structure intermédiaire
- C'est l'opération `MATRICE=COMBINAISON` qui déclenche le calcul en évitant la création de matrices intermédiaires
- Une seule recopie finale !

Un exemple plus évolué : implémentation

```
class MATRICE { //  classe MATRICE
  ...
};

class COMBINAISON{ // pointeurs matrice et coefficients
  public:
    int nb_matrice;
    MATRICE **liste_matrice;
    double *liste_coefficients;
    ...
}

COMBINAISON & operator*(double, const MATRICE &);
...
```

Attention au mécanisme de création/destruction des objets intermédiaires combinaison!

Méthodologie

- Ne jamais surcharger un opérateur avec un autre sens que le sens commun (`+` avec `-`)
- limiter le nombre d'implémentations pour des opérations similaires (`+` et `+=`)
- ne renvoyer des pointeurs qu'exceptionnellement, résultat inutilisable dans une autre opération :

  si `A+B` retourne un pointeur `(A+B)+C` est impossible
- pour une classe personnelle ne surcharger que l'essentiel
- pour une classe générale, prévoir toutes les surcharges possibles (complétude)
- la surcharge des opérateurs n'est pas obligatoire elle apporte un confort et augmente la lisibilité

Ce que toutes les classes doivent avoir

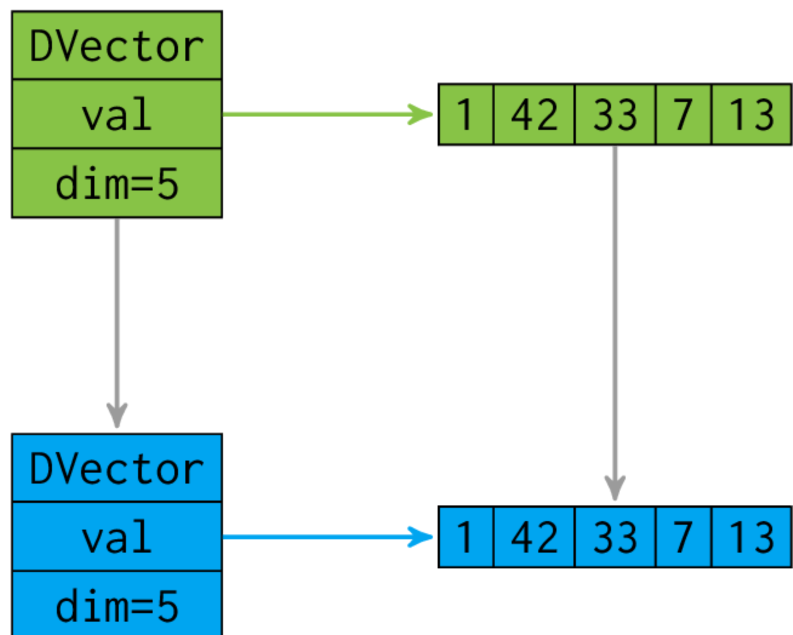
Règle de 3

- Si une classe définie l'une des 3 méthodes suivantes, alors les 3 doivent être définies
    1. Destructeur : `~Polynome()`.
    2. Constructeur par recopie : `Polynome(const Polynome &)`.
    3. Opérateur de recopie : `operator=(const Polynome &)`.
- Il est également important de re-définir un constructeur.

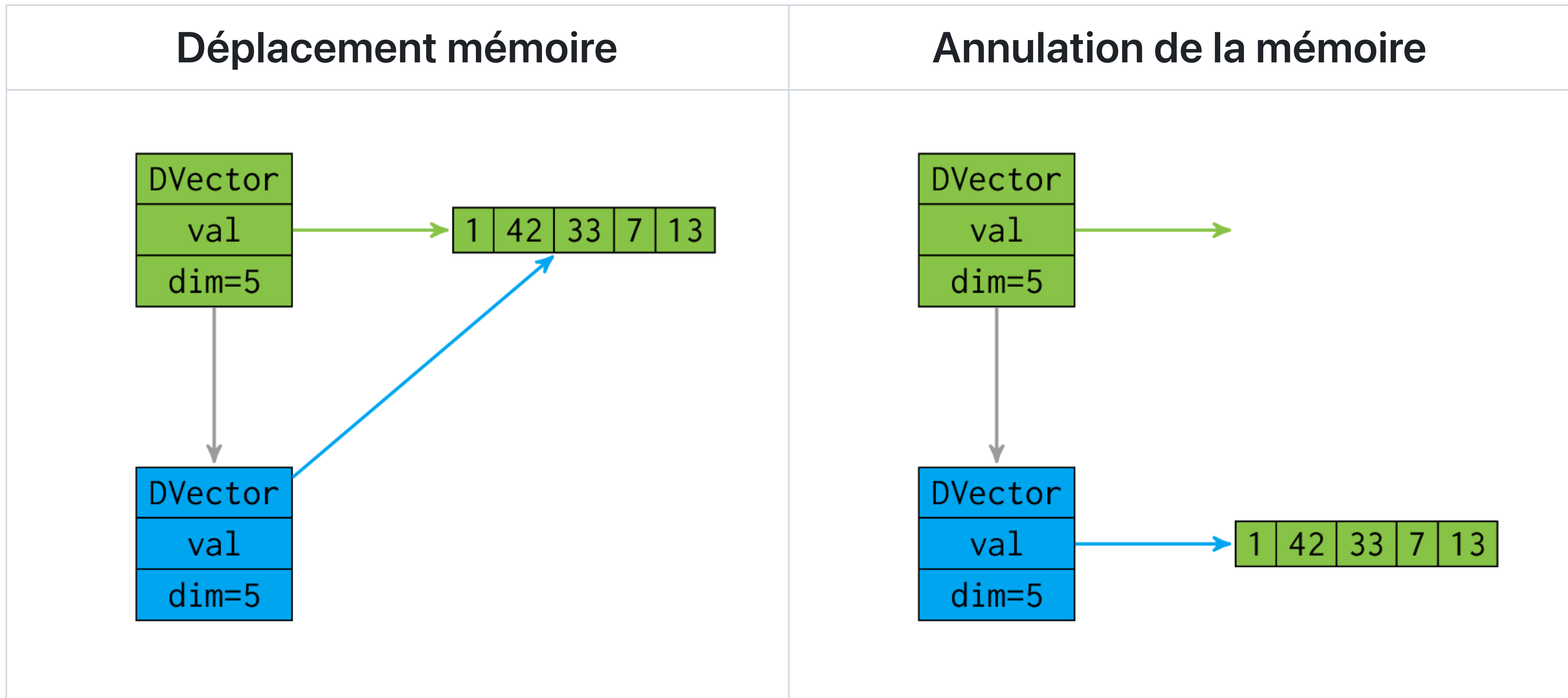
Règle de 5

- Avec le `C++ 2011`, il est recommandé de définir deux méthodes complémentaires
    1. Constructeur par déplacement mémoire : `Polynome(Polynome &&)`.
    2. Recopie par déplacement mémoire : `operator=(Polynome &&)`.
- En `C++ 2011`, il est possible de déléguer la création d'une de ces méthodes ou d'interdire son utilisation en utilisant des qualificateurs : `default` ou `delete`.

Sémantique de copie



Sémantique de déplacement



Conclusion

A retenir

- Surcharge d'opérateur.
- Règles de 3-5;

La prochaine fois

- Patterns