# Petite introduction à la Progammation orientée objet.

Thierry Dumont
Institut Camille Jordan, Lyon.

22 Mai 2012, Angers.

# Camille Jordan

- contrairement à ce que semblent penser les étudiants de Lyon 1, ce n'était pas un basketeur !

# Camille Jordan

- contrairement à ce que semblent penser les étudiants de Lyon 1, ce n'était pas un basketeur !
- J**O**rdan, pas J**OU**rdan, merci.

# Camille Jordan

- contrairement à ce que semblent penser les étudiants de Lyon 1, ce n'était pas un basketeur !
- J**O**rdan, pas J**OU**rdan, merci.

Né 5 Janvier 1838, à La Croix-Rousse, décès le 22 Janvier 1922,
Neveu de Pierre Puvis de Chavannes, peintre symboliste.
Son Grand Oncle : Ennemond-Camille Jordan : député royaliste. La rue Camille Jordan à Lyon lui est dédiée.

# Camille Jordan

- contrairement à ce que semblent penser les étudiants de Lyon 1, ce n'était pas un basketeur !
- J**O**rdan, pas J**OU**rdan, merci.

Né 5 Janvier 1838, à La Croix-Rousse, décès le 22 Janvier 1922,
Neveu de Pierre Puvis de Chavannes, peintre symboliste.
Son Grand Oncle : Ennemond-Camille Jordan : député royaliste. La rue Camille Jordan à Lyon lui est dédiée.

Le rève.

# Camille Jordan

- contrairement à ce que semblent penser les étudiants de Lyon 1, ce n'était pas un basketeur !
- J**O**rdan, pas J**OU**rdan, merci.

Né 5 Janvier 1838, à La Croix-Rousse, décès le 22 Janvier 1922,
Neveu de Pierre Puvis de Chavannes, peintre symboliste.
Son Grand Oncle : Ennemond-Camille Jordan : député royaliste. La rue Camille Jordan à Lyon lui est dédiée.

Le rève.

Mesure de Jordan, Courbe de Jordan, Forme normale de Jordan, Théorie des Groupes.

# Programmation dans les années 70

- mise en avant des structures de contrôle

# Programmation dans les années 70

- mise en avant des structures de contrôle
- 1973 : N. Wirth *Systematic Programming : An Introduction (Prentice-Hall Series in Automatic Computation)*, bible de l'époque.

**Algorithm + data structure = program.**

# Programmation dans les années 70

- mise en avant des structures de contrôle
- 1973 : N. Wirth *Systematic Programming : An Introduction (Prentice-Hall Series in Automatic Computation)*, bible de l'époque.

**Algorithm + data structure = program.**

Comment représenter des structures de données de plus en plus complexes ?
tableaux $\rightarrow$ graphes etc.

Algol (60, W, 68) Fortran : rien de plus que des tableaux.

Algol (60, W, 68) Fortran : rien de plus que des tableaux.

**C :** `struct` pour représenter des structures de données :

```
1 struct Array
2 {
3   int nb_columns,nb_ligns;
4   float *x;
5 };
```

Algol (60, W, 68) Fortran : rien de plus que des tableaux.

**C :** `struct` pour représenter des structures de données :

```
1  struct Array
2  {
3    int nb_columns,nb_ligns;
4    float *x;
5  };
```

```
1  Array Z;
```

**Comment initialiser Z ?**

```
1  Z.nb_columns=10; Z.nb_ligns=20;
2  Z.x=malloc(nb_columns*nb_ligns*sizeof(float));
```

Algol (60, W, 68) Fortran : rien de plus que des tableaux.

**C :** `struct` pour représenter des structures de données :

```
1  struct Array
2  {
3    int nb_columns,nb_ligns;
4    float *x;
5  };
```

```
1  Array Z;
```

**Comment initialiser Z ?**

```
1  Z.nb_columns=10; Z.nb_ligns=20;
2  Z.x=malloc(nb_columns*nb_ligns*sizeof(float));
```

**Comment manipuler Z ?**

```
1  Z[i,j]????
```

On peut stocker colonne après colonne : `Z[i,j]` :
`*(Z.x+j*Z.nb_columns+i);`
ou ligne après ligne : `Z[i,j]:` `*(Z.x+i*Z.nb_ligns+j);`

On peut passer Z à une procédure :

```
1 double sumcoeffs(Array X)
2 {
3    double ret=0.0;
4    for(int i=0;i<X.nb_columns*X.nb_ligns;i++)
5        ret+= X.x[i];
6    return ret
7 }
8 double s=sumcoeffs(Z);
```

- la représentation (= structure) des objets, les structures de données doivent être visibles partout

- la représentation (= structure) des objets, les structures de données doivent être visibles partout
- Les programmes sont rigides .
- crise de la programmation.

- la représentation (= structure) des objets, les structures de données doivent être visibles partout
- Les programmes sont rigides .
- crise de la programmation.

=> Découpler la représentation des objets des usages qu'on en fait.

L'abstraction rend fort => rendre les programmes plus abstraits.

- la représentation (= structure) des objets, les structures de données doivent être visibles partout
- Les programmes sont rigides .
- crise de la programmation.

=> Découpler la représentation des objets des usages qu'on en fait.

L'abstraction rend fort => rendre les programmes plus abstraits.

Langage historique : Simula 67 .

# Plan :

1. classes,
2. méthodes,
3. héritage et polymorphisme.

# Plan :

1. classes,
2. méthodes,
3. héritage et polymorphisme.

au delà :

1. généricité,
2. ... et quelques autres concepts.

# Plan :

1. classes,
2. méthodes,
3. héritage et polymorphisme.

au delà :

1. généricité,
2. ... et quelques autres concepts.

Langages ciblés : C++ et Python.

# Classes I

Cacher les structures de données.
Dans l'exemple précédent, on stocke un tableau *plein* .
Mais si je veux représenter un tableau creux $=>$ représentation parcimonieuse :
dictionnaire : $(i, j) \rightarrow \text{valeur}$.

# Classes

```
1 struct Array
2 {
3     int nb_columns,nb_ligns;
4     float *x;
```

# Classes

```
struct Array
{
  int nb_columns,nb_ligns;
  float *x;
```

```
  Array(int nbc,int nbl)//constructeur
  {
    nb_columns=nbc; nb_ligns=nbl;
    x=new float[nb_columns* nb_ligns];
  }
```

# Classes

```
struct Array
{
  int nb_columns,nb_ligns;
  float *x;
```

```
  Array(int nbc,int nbl)//constructeur
  {
    nb_columns=nbc; nb_ligns=nbl;
    x=new float[nb_columns* nb_ligns];
  }
```

```
  ~Array()//destructor
  {delete[] x;}
```

# Classes

```
1  struct Array
2  {
3    int nb_columns,nb_ligns;
4    float *x;
```

```
1    Array(int nbc,int nbl)//constructeur
2    {
3      nb_columns=nbc; nb_ligns=nbl;
4      x=new float[nb_columns* nb_ligns];
5    }
```

```
1    ~Array()//destructor
2    {delete[] x;}
```

```
1    float sumcoeffs() const
2    {
3      float ret=0;
4      for(int i=0;i< nb_columns*nb_ligns;i++)
5          ret+=x[i];
6      return ret;
```

# Classes

On a ajouté à la `struct` C des *méthodes* :

- un constructeur,
- un destructeur
- la méthode `sumcoeffs`.

# Classes

On a ajouté à la `struct` C des *méthodes* :

- un constructeur,
- un destructeur
- la méthode `sumcoeffs`.

Alors, on écrira :

```
1  Array X(20,25);
2  ....
3  float s=X.sumcoeffs();
```

# Classes

On a ajouté à la `struct` C des *méthodes* :

- un constructeur,
- un destructeur
- la méthode `sumcoeffs`.

Alors, on écrira :

```
1  Array X(20,25);
2  ....
3  float s=X.sumcoeffs();
```

**Encapsulation des données.**
**On a caché complètement la mécanique interne de Array.**

# Classes

On a ajouté à la `struct` C des *méthodes* :

- un constructeur,
- un destructeur
- la méthode `sumcoeffs`.

Alors, on écrira :

```
1  Array X(20,25);
2  ....
3  float s=X.sumcoeffs();
```

**Encapsulation des données.**
**On a caché complètement la mécanique interne de Array.**
**On peut changer complètement la représentation de Array sans conséquences sur le reste du programme.**

# Classes

En C++ :

```
1  class Array
2  {
3     int nb_columns,nb_ligns;
4     float *x;
5  public:
6     Array(int nbc,int nbl)//constructeur
7       {...
8       }
9     ....
10 };
```

# Classes

En C++ :

```
1  class Array
2  {
3     int nb_columns, nb_ligns;
4     float *x;
5  public:
6     Array(int nbc, int nbl)//constructeur
7       {...
8       }
9     ....
10 };
```

Parties `private`, `protected`, `public`.

# Classes

En C++ :

```
1  class Array
2  {
3     int nb_columns, nb_ligns;
4     float *x;
5  public:
6     Array(int nbc, int nbl)//constructeur
7       {...
8       }
9     ....
10 };
```

Parties `private`, `protected`, `public`.

```
1     Array X(20,25);
2     int n=X.nb_columns;
```

**ERREUR !**

# Classes

Et si je veux accéder au nombres de colonnes ?

# Classes

Et si je veux accéder au nombres de colonnes ?

Il faut ajouter une méthode dans la partie publique :

```
1 | int HowManyColumns() const {return nb_columns;}
```

# Classes

Et si je veux accéder au nombres de colonnes ?

Il faut ajouter une méthode dans la partie publique :

```
1 | int HowManyColumns() const {return nb_columns;}
```

Cette méthode renvoie une copie (sur la pile) de `nb_columns` et donc je ne pourrais pas modifier `nb_columns` :

**POO== sécurité.**

# Classes

En résumé (C++) :

```
1  class Array
2  {
3  private:
4  ......
5  protected:
6  .....
7  public:
8     Array(...){...}
9     ~Array(){...}
10   float sumcoeffs{...}
11    ....
12 };
```

## Questions :

- accès à un composant `(i,j)` ?

## Questions :

- accès à un composant `(i,j)` ?
- opérations entre `Array` ? `(+ - * /)` ?

## Questions :

- accès à un composant `(i,j)` ?
- opérations entre `Array` ? (`+ - * /`) ?
- Pourquoi réécrire `Array` si on veut faire des tableaux d'autres types d'objets ? (par exemple des `int` à la place des `float`).

# Classes en Python

```
1  class Personne:
2      """ bla bla, mais regardez mes doctests"""
3      def __init__(self,jour,mois,annee):
4          self.njour=jour
5          self.nmois=mois
6          self.nannee=annee
7      def age(self):
8          """ calculer l'age;  bidon ici"""
9          return 10
```

# Classes en Python

```
1  class Personne:
2      """ bla bla, mais regardez mes doctests"""
3      def __init__(self,jour,mois,annee):
4          self.njour=jour
5          self.nmois=mois
6          self.nannee=annee
7      def age(self):
8          """ calculer l'age;  bidon ici"""
9          return 10
```

```
1  x=Personne(2,4,3)
2  x.age()
3  10
```

# Classes en Python

- typé par le contexte $=>$ pas de déclarations de variables.
- pas de protection d'accès.
- un constructeur.
- pas de destructeur.

# Classes en Python

- typé par le contexte => pas de déclarations de variables.
- pas de protection d'accès.
- un constructeur.
- pas de destructeur.

C'est assez proche d'une classe C++ ou tout serait public (= une struct avec des méthodes) : l'abstraction n'est pas obligatoire :

```
1 x=Personne(2,4,3)
2 print x.njour
```

valide, mais pas très judicieux, viole le principe d'encapsulation des données .

*On est entre programmeurs adultes.*

# Héritage

On part d'une classe de base B .
On va créer des classes qui seront des B :

- auxquelles on aura rajouté des variables et des méthodes,
- pour lesquelles on aura redéfini des méthodes.

# Héritage

On part d'une classe de base B .
On va créer des classes qui seront des B :

- auxquelles on aura rajouté des variables et des méthodes,
- pour lesquelles on aura redéfini des méthodes.

**Buts :**

1. réutiliser du code existant.
2. manipuler ensemble des objets différents, mais partageant des propriétés communes.
3. ajouter de la sémantique au programme.

# Héritage : syntaxes

```
1  class Personne
2  {
3   protected:
4     int njour,nmois,nannee;
5  public:
6     Personne(int j,int n,int a)
7     {
8        njour=j; nmois=n; nannee=a;
9     }
10 .....
11 };
```

# Héritage : syntaxes

```
1 class AbonneAuGaz(Personne):
2     def __init__(self,jour,mois,annee,q):
3         self.quelque_chose=q
4         Personne.__init__(self,jour,mois,annee)
```

# Héritage : syntaxes

```
1 class AbonneAuGaz(Personne):
2     def __init__(self,jour,mois,annee,q):
3         self.quelque_chose=q
4         Personne.__init__(self,jour,mois,annee)
```

Dans ces deux langages (C++, Python) on peut avoir de l'héritage multiple.

# Héritage : redéfinir des méthodes I

Exemple : une classe de formes :

```
1  class Form
2  {
3  ....
4  public:
5     Form()
6     {
7
8     }
9  protected:
10    virtual void move(float dx,float dy)
11    {
12       ....
13    }
14
15 };
```

# Héritage : redéfinir des méthodes II

On a des objets de types différents, mais qui sont tous des formes : cercle, pentagone, segment...

# Héritage : redéfinir des méthodes : le polymorphisme.

```
1  class Segment: public Form
2    float x1,y1,x2,y2;
3  public:
4     Segment(float _x1,float _y1,float _x2,float _y2)
5       {
6         x1=_x1; x2=_x2; y1=_y1;y2=_y2;
7       }
8  virtual void move(float dx,float dy)
9       {
10        x1+=dx; y1+=dy;x2+=dx;y2+=dy;
11      }
12 };
```

Pour un Cercle (défini par son centre et son rayon, la méthode move sera différente.

# Héritage : redéfinir des méthodes

## Quel interet ?

Les Cercles, Segments etc... sont tous des Formes ; on peut les stocker dans un même conteneur :

```
1 Form F[3];
2 F[0]=cercle(1,2,0.5);
3 F[1]=Segment(1,2,2,-1);
4 F[2]=Triangle(1,2,2,3,3,4);
```

# Héritage : redéfinir des méthodes

## Quel interet ?

Les Cercles, Segments etc... sont tous des Formes ; on peut les stocker dans un même conteneur :

```
1 Form F[3];
2 F[0]=cercle(1,2,0.5);
3 F[1]=Segment(1,2,2,-1);
4 F[2]=Triangle(1,2,2,3,3,4);
```

Alors :

```
1 for(int i=0;i<3;i++)
2    Form[i].move(0.5,0.5);
```

# Héritage : ajouter de la sémantique

Exemples :

1. on veut empêcher de copier certains objets.
   On peut par exemple faire deriver la classe d'une classe qui déclanche une erreur en cas de copie :

```
1 class Toto: public noncopyable
2 {
3 ...
4 };
```

# Héritage : ajouter de la sémantique

Exemples :

1. on veut empêcher de copier certains objets.
   On peut par exemple faire deriver la classe d'une classe qui déclanche une erreur en cas de copie :

```
1 class Toto: public noncopyable
2 {
3 ...
4 };
```

2. Pour trier une collection d'objets, il faut que la collection permette l'accès aléatoire. On peut la faire dériver d'une classe (éventuellement vide) :

```
1 class Vecteur: public aleatoire
2  {
3
4 };
```

   Notion de concept dans la `stl`.

# Héritage : et en Python ?

Toutes les méthodes sont « virtuelles » : il suffit de réécrire les méthodes dans les classes dérivées.

```
class A:
    def ecrire_message(self):
        print "je suis un A."
```

```
class B(A):
    def ecrire_message(self):
        print "je suis un B."
```

# Héritage : et en Python ?

Toutes les méthodes sont « virtuelles » : il suffit de réécrire les méthodes dans les classes dérivées.

```
class A:
    def ecrire_message(self):
        print "je suis un A."
```

```
class B(A):
    def ecrire_message(self):
        print "je suis un B."
```

Python a de nombreuses formes d'introspection :

```
1 x=A
2 y=B
```

# Héritage : et en Python ?

Toutes les méthodes sont « virtuelles » : il suffit de réécrire les méthodes dans les classes dérivées.

```
class A:
    def ecrire_message(self):
        print "je suis un A."
```

```
class B(A):
    def ecrire_message(self):
        print "je suis un B."
```

Python a de nombreuses formes d'introspection :

```
1 x=A
2 y=B
```

```
1 isinstance(A,x)
2 True
3 isinstance(B,x)
4 False
5 isinstance(B,y)
6 True
7 isinstance(A,y)
8 True
```

# Au delà...

Reprenons notre classe de tableaux.

`Array X(20,10);`

J'aimerais bien accéder à $X_{i,j}$.

C++ et Python : surcharge d'opérateurs :

# Au delà...

Reprenons notre classe de tableaux.

`Array X(20,10);`

J'aimerais bien accéder à $X_{i,j}$.

C++ et Python : surcharge d'opérateurs :

On peut redéfinir les opérateurs habituels :

```
1 class Array{
2 .....
3 public:
4 .....
5 float operator()(int i,int j)
6    {.....}
7 };
```

# Au delà...

Reprenons notre classe de tableaux.

`Array X(20,10);`

J'aimerais bien accéder à $X_{i,j}$.

C++ et Python : surcharge d'opérateurs :

On peut redéfinir les opérateurs habituels :

```
1 class Array{
2 .....
3 public:
4 .....
5 float operator()(int i,int j)
6    {.....}
7 };
```

et en Python il faut créer des méthodes `__add__`, `__mul__`, pour définir l'addition, la multiplication etc...

# Au delà...

```
1 class Complex:
2     def __init__(self,a,b):
3         self.a=a
4         self.b=b
5     def __add__(self,C):
6         return Complex(self.a+C.a,self.b+C.b)
```

# Au delà...

```
class Complex:
    def __init__(self,a,b):
        self.a=a
        self.b=b
    def __add__(self,C):
        return Complex(self.a+C.a,self.b+C.b)
```

```
x=Complex(1.,2.)
y=Complex(2.,3.)
c=x+y
c.a
3
c.b
5
```

Bon, ok,.

# Au delà...

```
1 x=Complex(``truc'',''machin'')
2 y=Complex(``toto'',''tutu'')
3 c=x+y
```

Ça va marcher car “+” existe entre chaînes de caractères.

# Au delà...

```
1 x=Complex(``truc'',''machin'')
2 y=Complex(``toto'',''tutu'')
3 c=x+y
```

Ça va marcher car “+” existe entre chaînes de caractères.

```
1 c.a
2 tructoto
3 c.b
4 machintutu
```

# Au delà...

```
1 x=Complex(``truc'',''machin'')
2 y=Complex(``toto'',''tutu'')
3 c=x+y
```

Ça va marcher car "+" existe entre chaînes de caractères.

```
1 c.a
2 tructoto
3 c.b
4 machintutu
```

**Généricité.**

*voir l'exemple en ligne.*

# Au delà...

```
1 x=Complex(``truc'',''machin'')
2 y=Complex(``toto'',''tutu'')
3 c=x+y
```

Ça va marcher car "+" existe entre chaînes de caractères.

```
1 c.a
2 tructoto
3 c.b
4 machintutu
```

**Généricité.**

*voir l'exemple en ligne.*

A partir du moment ou toutes opérations sont définies, les procédures, classes etc. sont complètement réutilisables.

# Généricité

Python : langage interpreté => relativement facile.

C++ : `template`. C'est statique, à la compilation.

# Généricité

Python : langage interpreté => relativement facile.
C++ : `template`. C'est statique, à la compilation.

```
1  template<class T> class Array
2   {
3      T* x;
4      int nx,ny;
5  public:
6      Array(int _nx,int _ny): nx(_nx),ny(_ny)
7       {
8          x=new T[nx*ny];
9       }
10 ....
11 };
```

# Généricité

Python : langage interpreté => relativement facile.

C++ : `template`. C'est statique, à la compilation.

```
template<class T> class Array
 {
    T* x;
    int nx,ny;
public:
    Array(int _nx,int _ny): nx(_nx),ny(_ny)
     {
        x=new T[nx*ny];
     }
....
};
```

```
Array<float> X(10,20);
Array<int> Y(12,4);
Array<ofstream> Q(2,2);
```

# Généricité

```
1  template<class T> class Array
2   {
3  ...
4     void operator+=(Array<T>& Z)
5      {
6        for(int i=0;i<nx*ny;i++)
7           x[i]+=Z.x[i];
8      }
9  ...
```

# Généricité

```
1  template<class T> class Array
2   {
3  ...
4     void operator+=(Array<T>& Z)
5      {
6         for(int i=0;i<nx*ny;i++)
7             x[i]+=Z.x[i];
8      }
9  ...
```

Ne sera « compilable »que si « += »existe entre objets de type T.

# Généricité

```
1  template<class T> class Array
2   {
3  ...
4     void operator+=(Array<T>& Z)
5      {
6        for(int i=0;i<nx*ny;i++)
7            x[i]+=Z.x[i];
8      }
9  ...
```

Ne sera « compilable »que si « += »existe entre objets de type `T`. Alors, on pourra écrire :

```
1     Array<int> X(10,10);
2     Array<int> Y(10,10);
3  ....
4     X+=Y;
```

l'opérateur « += »peut tester si les tableaux ont la même taille.

# Généricité

**Une idée géniale et fondatrice : la STL C++** (Oleg Stepanov et Ang Lee).

Bibliothèque de (templates de) :

- conteneurs,
- adaptateurs,
- itérateurs,
- algorithms.

# Généricité

**Une idée géniale et fondatrice : la STL C++** (Oleg Stepanov et Ang Lee).

Bibliothèque de (templates de) :

- conteneurs,
- adaptateurs,
- itérateurs,
- algorithms.

```
1 Vector<T>
2 map<T,Q>
3 set<T>
4 list<T>
```

Certains conteneurs (`set`, `map`, ...) ne peuvent contenir que des objets ordonnés (l'opérateur « <= »peut être surchargé).

# Généricité

**Une idée géniale et fondatrice : la STL C++** (Oleg Stepanov et Ang Lee).

Bibliothèque de (templates de) :

- conteneurs,
- adaptateurs,
- itérateurs,
- algorithms.

```
1 Vector<T>
2 map<T,Q>
3 set<T>
4 list<T>
```

Certains conteneurs (`set`, `map`, ...) ne peuvent contenir que des objets ordonnés (l'opérateur « <= »peut être surchargé).

Exemple :

```
1 map<pair<int,int>,float>
```

$(i, j) \to x$

(B-trees : accès en $O(\log_2 n)$).

# Généricité, STL.

Adaptateur : à partir de certains conteneurs on construit des piles, des FIFO etc...

# Généricité, STL.

Adaptateur : à partir de certains conteneurs on construit des piles, des FIFO etc...

Algorithm

Exemple : le tri (c'est un template de fonction) :

```
1 template<class T> void sort(T& X)
```

- `T` doit être un conteneur qui possède l'accès aléatoire (ok pour les `vector`, pas pour les `queue`) : propriété codée dans l'héritage,

# Généricité, STL.

Adaptateur : à partir de certains conteneurs on construit des piles, des FIFO etc...

Algorithm

Exemple : le tri (c'est un template de fonction) :

```
1 template<class T> void sort(T& X)
```

- `T` doit être un conteneur qui possède l'accès aléatoire (ok pour les `vector`, pas pour les `queue`) : propriété codée dans l'héritage,
- `T` doit contenir des objets ordonnés.

```
1 vector<int> X;
2 vector<pair<int,float>> Y;
3 ....
4 X.sort(); Y.sort();
```

# Généricité, STL.

## Iterateurs.

Indispensable pour écrire des programmes.

```
1
2  template<class C> float sumcoeffs(const C& X)
3   {
4      float s=0.;
5      for(C::iterator I=X.begin();I!=X.end();I++)
6          s+=I->second;
7      return s;
8   }
```

# Généricité, STL.

## Iterateurs.

Indispensable pour écrire des programmes.

```
1
2  template<class C> float sumcoeffs(const C& X)
3   {
4      float s=0.;
5      for(C::iterator I=X.begin();I!=X.end();I++)
6          s+=I->second;
7      return s;
8   }
```

```
1  map<pair<int,int>,float> X;
2  ....
3  float s=sumcoeffs(X);
```

(là, j'arnaque un peu).

# Généricité.

**Limitations !**

Il n'y a pas de sémantique associée aux opérateurs surchargés !

Exemples :

- Le langage ne sait pas si `a*(b+c) == a*b+a*c` !

# Généricité.

**Limitations !**

Il n'y a pas de sémantique associée aux opérateurs surchargés !

Exemples :

- Le langage ne sait pas si `a*(b+c) == a*b+a*c` !
- Reprenons l'exemple Python :

```
1     def __add__(self,C):
2         return Complex(self.a+C.a, self.b+C.b)
3
4 x=y+z
```

copies !

En C++, le remède s'appelle "expression templates" :

`http://en.wikipedia.org/wiki/Expression_templates`

Bonne lecture !☺

# Généricité.

**Limitations !**

Il n'y a pas de sémantique associée aux opérateurs surchargés !

Exemples :

- Le langage ne sait pas si `a*(b+c) == a*b+a*c` !
- Reprenons l'exemple Python :

```
1     def __add__(self,C):
2         return Complex(self.a+C.a,self.b+C.b)
3
4 x=y+z
```

  copies !

  En C++, le remède s'appelle "expression templates" :

  `http://en.wikipedia.org/wiki/Expression_templates`

  Bonne lecture !☺

En C++, plus fort que la STL : Boost `http://www.boost.org`.

# Contraindre, encore contraindre

## Classes abstraites

Contraindre les classes dérivées à implanter des fonctionnalités

# Contraindre, encore contraindre

## Classes abstraites

Contraindre les classes dérivées à implanter des fonctionnalités

```
1  class Form
2  {
3  ....
4  public:
5     Form()
6     {
7     }
8  protected:
9     virtual void move(float dx,float dy)=0;
10 };
```

# Contraindre, encore contraindre

## Classes abstraites

Contraindre les classes dérivées à implanter des fonctionnalités

```
1  class Form
2  {
3  ....
4  public:
5     Form()
6     {
7     }
8  protected:
9     virtual void move(float dx,float dy)=0;
10 };
```

Existe aussi maintenant en Python.

# Contraindre, encore contraindre

Python :

- Toutes les classes dérivent de façon cachée d'une classe de base,

# Contraindre, encore contraindre

Python :

- Toutes les classes dérivent de façon cachée d'une classe de base,
- Les classes sont elles-mêmes des objets : on peut créer des métaclasses (changer la classe de base).

# La boite de Pandore

Martyrisez votre compilateur...

# La boite de Pandore

Martyrisez votre compilateur...

```
1  template <int N>
2  struct Factorial {
3      enum { value = N * Factorial<N - 1>::value };
4  };
5
6  template <>
7  struct Factorial<0> {
8      enum { value = 1 };
9  };
10
11 // Factorial<4>::value == 24
12 // Factorial<0>::value == 1
13 const int x = Factorial<4>::value; // == 24
14 const int y = Factorial<0>::value; // == 1
```

# La boite de Pandore

Mal au crane ? Fatigué(e) ?

# La boite de Pandore

Mal au crane ? Fatigué(e) ?

**NON ? ? ?**

# La boite de Pandore

Mal au crane ? Fatigué(e) ?

**NON ? ? ?**

Alors essayez de comprendre l'astuce de Barton et Nackman
`http://en.wikipedia.org/wiki/Template_metaprogramming`

# La boite de Pandore

```
1  template <class Derived>
2  struct base
3  {
4     void interface()
5     {
6     // ...
7     static_cast<Derived*>(this)->implementation();
8     // ...
9     }
10 };
11
12 struct derived : base<derived>
13 {
14    void implementation();
15 };
```