Programmation C++

Quelques rappels

Pierre Ramet

pierre.ramet@inria.fr

Stage Programmation Licence, 2019

IUT Bordeaux

Plan de l'exposé

Généralités sur le C++

Allocation statique

Passage de paramètres

Constructeur par copie ; affectation ; surcharge

Pointeurs ; allocation dynamique

Héritage ; polymorphisme

Plan de l'exposé

Généralités sur le C++

Allocation statique

Passage de paramètres

Constructeur par copie ; affectation ; surcharge

Pointeurs ; allocation dynamique

Héritage ; polymorphisme

Généralités sur le C++ i

- Programmation orientée objet : mélange impératif/objet
- Forte compatibilité avec le C, langage assez proche du langage machine.
- Classes : structures C enrichies avec des fonctions membres
- Possibilité d'héritage, de surcharge, de polymorphisme, de composition de classes
- Mécanisme d'exceptions

Généralités sur le C++ ii

- Une bibliothèque standard pour les interactions avec le système (provenant du C), plus la STL (standard template library) qui fournit quelques structures de données classiques.
- Beaucoup d'autres possibilités : patrons de classes, héritage multiple, allocation statique et dynamique, surcharge d'opérateurs, pointeurs et accès mémoire, préprocesseurs et macros

$\boxed{\Rightarrow}$ Le C++ est un langage un peu plus complexe à maîtriser que le JAVA : temps de développement plus long, maintenance plus délicate. Néanmoins les performances sont nettement meilleures.

Programme C++ i

- un programme C++ est découpé en fichiers sources et fichiers en-tête (*headers*).
  - les headers ont une extension .h, .H, .hpp : ils contiennent les déclarations des types/classes/structures et des fonctions et méthodes.
  - les sources ont une extension .cc, .C, .c++, .cpp, .cxx. Ils contiennent effectivement les instructions C++ associées aux fonctions et méthodes définies dans les headers, ainsi que les éventuelles variables globales.
  - en général, chaque fichier source a son fichier header qui lui correspond.

Programme C++ ii

- Chaque source doit ensuite être compilé en fichiers .o (sous Linux, g++ -c *source*.cc). Ces fichiers objets contiennent une table de symboles et le code en langage machine.
- Tous ces fichiers .o sont ensuite reliés ensemble (édition des liens) et avec des bibliothèques extérieures dans certains cas pour former un fichier, directement exécutable par le processeur.
- L'un des sources contient une fonction `main` appelé en premier par le système à l'exécution.

Premier exemple i

Regardez le code de `Echecs` et exécutez-le.

Premier exemple ii

```
// Fichier Piece.h
class Piece
{
private:
  int m_x;
  int m_y;
  bool m_white;

public:
  Piece(); Constructeur
  void init( int x, int y, bool white );
  void move( int x, int y );
  int x();
  int y();
  bool isWhite();
};
```

Premier exemple i

1. Observez qu'une classe est définie dans un fichier en-tête, afin d'être utilisable par d'autres modules.
2. Constructeurs

   Faites un nouveau constructeur qui prend en paramètres x,y et la couleur. Instanciez un nouvel objet p2 et affichez-le.

3. Sectionnement `public`/`private`

   Essayez d'accéder à l'attribut `m_x` directement dans le `main` en écrivant `p1.m_x = 2;` . Peut-on toujours compiler ? Mettez-le en section `public`. Que se passe-t-il alors ?

Premier exemple ii

4. Méthodes

   Ecrivez une nouvelle méthode `isBlack` qui retourne l'inverse de `isWhite`.

5. Utilisation d'autres modules.

   On veut intégrer une méthode d'affichage dans la classe `Piece`, appelée `affiche`. Constatez qu'il faut alors include `<iostream>` dans `Piece.cxx`. Pourquoi ? Dans le programme principal, appelez plutôt la méthode `affiche`.

Plan de l'exposé

Généralités sur le C++

Allocation statique

Passage de paramètres

Constructeur par copie ; affectation ; surcharge

Pointeurs ; allocation dynamique

Héritage ; polymorphisme

Allocation statique i

- En C++ les variables déclarées sous la forme *type nom;* ont une durée de vie limitée à la fonction ou méthode où elles sont déclarées. On parle d'allocation statique.
- Constructeurs ; destructeur ; durée de vie d'un objet

Dans chaque constructeur de `Piece`, écrivez une ligne du genre `cout « "Une piece creee" « endl;`
Ecrivez une méthode de prototype `~Piece()` dans `Piece`. Faîtes que le corps de cette méthode affiche "Une piece detruite". Qu'en déduisez-vous sur la durée des objets `p1` et `p2` ? On dit que ces objets ont été alloués statiquement.

Allocation statique ii

- Tableau d'objets alloué statiquement. On écrit *type* *nom*`[nb];`

```
Piece tbl[ 4 ];
```

Dans le programme principal, déclarez un tableau de 4 pièces. Que constatez-vous ? Combien d'instances sont créées ? Est-ce similaire à JAVA ? Quel est le constructeur appelé ?

Allocation statique iii

Ecrire maintenant une classe `Joueur`. Cette classe représente un joueur d'échecs, qui joue les blancs ou les noirs. Il possède 16 pièces à des positions bien déterminées (coordonnée y : 1 ou 2 pour le blanc, 7 ou 8 pour le noir ; coordonnée x de 1 à 8 pour les deux). On ignorera pour le moment que les pièces sont différenciées. On proposera un constructeur de `Joueur` qui aura un comportement différent selon que le joueur est blanc ou noir. On testera cette classe en instanciant deux joueurs dans le programme principal. On écrira aussi une méthode affiche qui liste les pièces du joueur.

Allocation statique iv

- Vous constatez que l'instanciation d'un `Joueur` provoque l'instanciation de 16 pièces. Sa destruction provoque la destruction des 16 pièces.

Comment vérifier dans quel ordre sont construits et détruits les attributs de l'objet par rapport à l'objet lui-même ?

Plan de l'exposé

Généralités sur le C++

Allocation statique

Passage de paramètres

Constructeur par copie ; affectation ; surcharge

Pointeurs ; allocation dynamique

Héritage ; polymorphisme

Passage de paramètres i

- Lorsqu'on définit une fonction ou méthode avec des arguments, le passage des arguments peut se faire par valeur ou par référence.
- Passage par valeur : `fct( type nom )`
  Le paramètre formel est alors une copie de l'argument d'appel.

Ecrire une fonction ou méthode qui teste si deux pièces sont au même endroit. Vérifiez-la en comparant deux pièces quelconques des joueurs. Y a-t-il de nouvelles instances de `Piece` créées ?

Passage de paramètres ii

- Passage par référence : `fct( type & nom )`
  Le paramètre formel référence l'argument d'appel. Aucune nouvelle instance n'est créée. Toute modification de la référence modifie l'argument.

Réécrire la fonction ou méthode précédente avec un passage par référence.

- Le passage par pointeur des adeptes du C est en fait un passage par valeur d'un type qui est un pointeur.

Plan de l'exposé

Généralités sur le C++

Allocation statique

Passage de paramètres

Constructeur par copie ; affectation ; surcharge

Pointeurs ; allocation dynamique

Héritage ; polymorphisme

Constructeur par copie

Prototype

```
Piece::Piece(const Piece & autre);
```

Le constructeur de copie est appelé lors de la déclaration suivante :

```
Piece p1(1,1,true);
Piece p = p1;
```

et aussi lors du retour d'une `Piece` dans une fonction.

Opérateur d'affectation

- Définition implicite par le compilateur (comme le constructeur par défaut ou le constructeur de copie).
- Il est appelé pour une affectation avec un objet de la classe dans la partie gauche.

Prototype

```
Piece & operator=( const Piece & autre );
```

exemple

```
Piece p; // constructeur par defaut
Piece p2( 3, 3, true); // constructeur
p = p2; // operateur d'affectation
Piece p3 = p2; // constructeur par copie
Piece p3( p2 ); // equivalent
Piece p3 = Piece( p2 ); // equivalent
```

Exercice

Ecrivez une fonction membre de `Piece` qui retourne la pièce la plus forte entre 2 pièces. [quelques rappels sur `const`]

Réponse

```
Piece Piece::plusforte(const Piece & p) const {
  if (...) return p
  else return *this;
}
```

Remarque

Le constructeur par copie est automatiquement appelé pour récupérer le retour. Il est cependant possible de retourner une référence `const Piece&` pour éviter la recopie.

Exercice

Ecrivez une fonction membre de `Piece` qui retourne la pièce la plus forte entre 2 pièces. [quelques rappels sur `const`]

Réponse

```
Piece Piece::plusforte(const Piece & p) const {
  if (...) return p
  else return *this;
}
```

Remarque

Le constructeur par copie est automatiquement appelé pour récupérer le retour. Il est cependant possible de retourner une référence `const Piece&` pour éviter la recopie.

Exercice

Ecrivez une fonction membre de `Piece` qui retourne la pièce la plus forte entre 2 pièces. [quelques rappels sur `const`]

Réponse

```
Piece Piece::plusforte(const Piece & p) const {
  if (...) return p
  else return *this;
}
```

Remarque

Le constructeur par copie est automatiquement appelé pour récupérer le retour. Il est cependant possible de retourner une référence `const Piece&` pour éviter la recopie.

Retour de fonction par référence

On ne peut, par contre, absolument pas écrire :

```
const Piece & Piece::plusfort(const Piece & p) const {
  Piece tmp;
  ...
  return tmp;
}
```

car on retourne une référence sur du vide dans ce cas...

Solution avec des pointeurs (penser à faire `delete` !)

```
Piece* Piece::plusfort(const Piece & p) const {
  Piece* tmp = new Piece;
  ...
  return tmp;
}
```

Retour de fonction par référence

On ne peut, par contre, absolument pas écrire :

```
const Piece & Piece::plusfort(const Piece & p) const {
  Piece tmp;
  ...
  return tmp;
}
```

car on retourne une référence sur du vide dans ce cas...

Solution avec des pointeurs (penser à faire `delete` !)

```
Piece* Piece::plusfort(const Piece & p) const {
  Piece* tmp = new Piece;
  ...
  return tmp;
}
```

Surcharge

- Il est possible en C++ de surcharger le comportement des principaux opérateurs (+,-,*,/,<,>,==, ...).
- Il est également possible de surcharger l'accesseur d'un tableau.

Exemple pour une classe `Tableau`

```
float & Tableau::operator[](int i) const {
  return m_tab[i-1];
}
```

La surcharge des fonctions n'est pas réservée aux classes. Il peut être utile de surcharger des fonctions non membres.

Plan de l'exposé

Généralités sur le C++

Allocation statique

Passage de paramètres

Constructeur par copie ; affectation ; surcharge

Pointeurs ; allocation dynamique

Héritage ; polymorphisme

Pointeurs i

- Un pointeur de type `T` se définit avec `T* nom_pointeur`.
  Il permet de mémoriser l'adresse en mémoire d'un objet ou variable de type `T`.
- L'objet pointé par un pointeur peut être obtenu avec l'opérateur *
- On obtient l'adresse d'un objet ou variable avec l'opérateur &
- On accède à méthode/attribut d'objet pointé par la notation ->

Pointeurs ii

```
Piece tbl[ 4 ];
Piece* ptr = 0;    // Pointeur null
ptr = &tbl[ 0 ]; // ptr pointe vers tbl[ 0 ]
(*ptr).affiche(); // Affiche la piece tbl[ 0 ]
ptr->affiche();    // Idem
ptr++;             // ptr pointe vers tbl[ 1 ]
```

Allocation dynamique i

- Il est parfois nécessaire de créer des objets dont la durée de vie dépasse la portée locale. En C++, cela se fait par allocation dynamique, qui demande au système de réserver une zone mémoire.
- Pour créer une instance, on utilise l'opérateur **new**. Pour la détruire, on utilisera l'opérateur **delete**.
- l'opération `new T` retourne un pointeur de T. C'est donc au travers de pointeurs que l'on manipulera les objets créés dynamiquement.

Allocation dynamique ii

- lorsqu'on n'a plus besoin de cet objet, on le détruit explicitement avec `delete` *pointeur*.

```
Piece* allouePiece( int x, int y, bool white )
{
  Piece* ptr = new Piece( x, y, white );
  return ptr;
}
...
   Piece* ptr2 = allouePiece( 3, 4, false );
   ptr2->affiche();
   delete ptr2;
...
```

Allocation dynamique iii

- Les tableaux peuvent être aussi alloués ou désalloués dynamiquement par les commandes **new** T**[** nb **]** et **delete[]**. Attention l'allocateur retourne encore un pointeur de T pointant vers le premier élément

Exercices sur les pointeurs i

Récupérez les fichiers `Echiquier.h` et `Echiquier.cxx`. Pourquoi son attribut `m_cases` ne mémorise que des pointeurs et pas des pieces directement ? Ecrire proprement le constructeur pour que l'échiquier soit vide au début. Complétez les méthodes spécifiées. Testez en instanciant un échiquier et en l'affichant.

Exercices sur les pointeurs i

Ecrire ensuite une méthode de `Joueur` qui place automatiquement toutes ses pièces sur un échiquier donné en paramètre. Vérifiez le placement des pièces sur l'échiquier en l'affichant.

De la même manière, écrire une méthode de `Piece` :

```
bool Piece::mouvementValide(Echiquier &e, int x, int y);
```

[pb de référence croisée]

Différence entre Pointeur et Référence i

- Un pointeur peut parfaitement pointer vers une adresse invalide (NULL)
- Une référence doit être directement initialisée avec l'objet auquel elle fait référence

```
int * ptr;
/* ... */
ptr = new int;
```

```
int & ref; // interdit
/* ... */
ref = i;
```

Différence entre Pointeur et Référence ii

Un pointeur peut pointer vers un objet différent de celui d'origine

```
int i = 4;
int j = 6;
int *ptr = &i; // 'ptr' contient l'adresse memoire de 'i'
(*ptr)*=2;
cout << i << endl; // affiche 8
ptr = &j; // 'ptr' contient l'adresse memoire de 'j'
(*ptr)*=2;
cout << i << " " << j << endl; // affiche 8 12
```

Différence entre Pointeur et Référence iii

Il est impossible de changer l'objet référencé par une référence

```
int i = 4;
int j = 6;
int &ref = i; // 'ref' fait reference a 'i'
ref*=2;
cout << i << endl; // affiche 8
ref = j; // affecte la valeur de 'j' a 'ref' (idem i=j;)
cout << i << " " << j << endl; // affiche 6 6
i*=2;
cout << i << " " << j << endl; // affiche 12 6
```

La règle des trois grands i

Le constructeur par copie, l'opérateur d'affectation et le destructueur sont liés

Si, parmi les trois fonctions importantes que sont le constructeur par copie, l'opérateur d'affectation et le destructueur, il y en a une dont le comportement doit être redéfini (dont vous ne pouvez pas vous contenter d'utiliser le comportement par défaut implémenté par le compilateur), alors il faudra définir les trois !

La règle des trois grands ii

Exemple : `Matrix.h`

```
class Matrix {
public:
    Matrix(int l, int c);
    ~Matrix();
    int operator()(int l, int c) const;
private:
    int m_l;
    int m_c;
    int *m_tab;
};
```

La règle des trois grands iii

Exemple : `Matrix.cxx`

```
#include "Matrix.h"

Matrix::Matrix(int l, int c): m_l(l),m_c(c),
    m_tab(new int[l*c]) {}

Matrix::~Matrix() { delete [] m_tab; }

int Matrix::operator()(int l, int c) const {
    return m_tab[l*m_c+c];
}
```

La règle des trois grands iv

Utilisation qui semble fonctionner

```
#include "Matrix.h"

int main () {
    Matrix mat(10,10);
    return 0;
}
// Matrice::~Matrice() est appelee
// delete [] m_tab est invoque
// pas de fuite memoire
```

La règle des trois grands v

Utilisation qui pose problème

```
#include "Matrix.h"

int main () {
    Matrix mat(10,10); // une premiere matrice
    Matrix autre(mat); // BUG : une copie de 'mat'
    return 0;
}
// tentative de double liberation de la memoire !!!
```

De même, une affectation `mat=autre;` posera également problème !

La règle des trois grands vi

Ajout du constructeur par copie et de l'opérateur d'affectation

```
Matrix::Matrix(Matrix const & autre):
    m_l(autre.m_l),m_c(autre.m_c),
    m_tab(new int[autre.m_l*autre.m_c]) {
    memcpy(m_tab,autre.m_tab,(m_l*m_c)*sizeof(int));
}

Matrix & Matrix::operator=(Matrix const & autre) {
    assert(m_l==autre.m_l && m_c==autre.m_c);
    memcpy(m_tab,autre.m_tab,(m_l*m_c)*sizeof(int));
    return *this;
}
```

Plan de l'exposé

Généralités sur le C++

Allocation statique

Passage de paramètres

Constructeur par copie ; affectation ; surcharge

Pointeurs ; allocation dynamique

Héritage ; polymorphisme

Héritage i

- Le langage C++ permet l'héritage entre classes, même multiple. On écrit

```
class B :  accesseur A
{ // B herite de A
};
```

  accesseur est soit `public`, `protected` ou `private`.
- une instance de B est alors aussi une instance de A.
- Dans le cas d'un héritage public, B a accès à tous les attributs et méthodes publics de A.
- B a accès à tous les attributs et méthodes `protected` de A.

Héritage ii

Définissez deux classes `JoueurBlanc` et `JoueurNoir` qui héritent de `Joueur` et dont les constructeurs initialisent correctement les pièces. Mettez à jour `Joueur` en éliminant le constructeur `Joueur( bool )`. Mettez du texte dans le constructeur par défaut de `Joueur` et dans les destructeurs de ces classes pour observer dans quel ordre sont appelés les constructeurs à l'instanciation et les destructeurs à la fin du programme.
Au lieu d'instancier deux joueurs, instanciez un `JoueurBlanc` et un `JoueurNoir`. Vérifiez que vous pouvez toujours les afficher avec `affiche`.

Polymorphisme ; méthodes virtuelles i

- Polymorphisme : traiter plusieurs formes d'une classe comme si elles n'en étaient qu'une, sans perte de leurs spécificités
- Contrairement au JAVA, en C++ il faut spécifier explicitement quelles vont être les méthodes polymorphes avec **virtual**.

```
class Piece {
  void f() {
    cout « "Piece::f" « endl;
  }
}
```

```
class Piece {
  virtual void f() {
    cout « "Piece::f" « endl;
  }
}
```

Polymorphisme ; méthodes virtuelles ii

```
class Roi : public Piece {
  void f() {
    cout « "Roi::f" « endl;
  }
}
...
  Roi r; // Cree une instance de Roi
  Piece* ptr = &r; // Cree un pointeur de type Piece vers r
  r.f(); // Roi::f
  ptr->f(); // Piece::f ou (virtual) Roi::f
```

Polymorphisme ; méthodes virtuelles iii

- Attention : si une classe contient une méthode virtuelle, il est très conseillé de mettre son destructeur virtuel aussi.

```
// Piece.h
class Piece {
  virtual ~Piece();
  virtual void f();
}
```

```
// Piece.cxx
Piece::~Piece() { ... }
Piece::f() { ... }
```

- Un constructeur ne peut être virtuel.

Exercice

On va pouvoir modéliser maintenant chaque pièce du jeu d'échecs avec son comportement propre. On écrira donc une classe par type de pièce : `Roi`, `Reine`, `Fou`, `Cavalier`, `Tour`, `Pion`. Ces classes spécialiseront une méthode virtuelle de pièce de prototype :

```
bool mouvementValide( Echiquier & e, int x, int y );
```

Attention, certaines pièces ont des mouvements autorisés différents selon noir ou blanc. Pourquoi passer un echiquier en paramètre ? Vous pourrez aussi spécialiser l'affichage des pièces ainsi : Blanc : RQFCTP, Noir : rqfctp. Dans les joueurs, instanciez correctement les nouvelles pièces.

Diminuer la visibilité d'une fonction membre

```
class Base {
public:
    virtual ~Base(){}
    virtual void doSomething() {
        cout << "fonction publique" << endl; }
};
class Derivee : public Base {
private:
    virtual void doSomething() {
        cout << "fonction usage interne" << endl; }
};
void bar(Base &b) { b.doSomething(); }
int main () {
    Derivee d;
    bar(d); // affiche fonction usage interne !!!
}
```

Augmenter la visibilité d'une fonction membre

```
class Base {
public:
    virtual ~Base(){}
protected:
    virtual void doSomething() {
        cout << "fonction protegee" << endl; }
};
class Derivee : public Base {
public:
    virtual void doSomething() {
        cout << "fonction publique" << endl; }
};
void bar(Base &b) { b.doSomething(); }
// error: 'doSomething' is a protected member of 'Base'
```

Polymorphisme ; Transtypage i

- Un pointeur de type `T` peut pointer vers tout objet de type `T'` où `T'` est une spécialisation de `T`. Il est parfois utile de retrouver le type réel d'un objet pointé, par exemple pour appeler des méthodes spécialisées qui ne sont pas définies dans le type de base.
- On voudrait, à partir d'un pointeur sur une `Piece` pointant vers un "roi" par exemple, obtenir un pointeur sur un `Roi` sur cette même instance. On parle de *transtypage*.

Polymorphisme ; Transtypage ii

```
Roi rb(true);
Piece* ptr = &rb;
Roi* rptr = ptr; // interdit
```

En effet, le compilateur ne peut pas autoriser une telle instruction car rien ne dit que l'élément pointé par `ptr` a été effectivement instancié comme un objet `Roi`. Lorsque l'on souhaite transtyper un pointeur, il faut donc l'indiquer explicitement. Cela se fait avec l'opérateur `dynamic_cast`.

Polymorphisme ; Transtypage iii

```
Roi rb(true);
Piece* ptr = &rb;
Roi* rptr = dynamic_cast<Roi*>( ptr ); // ok
if ( rptr == 0 )
  cerr « "ptr ne pointait pas sur une instance de Roi." « endl;
else ...
```

L'opérateur `dynamic_cast` analyse le type réel de l'objet pointé et, si le transtypage est valide, retourne un pointeur du bon type, sinon il retourne le pointeur `0`.

Polymorphisme ; Transtypage iv

Dans notre exemple, la méthode `roque` a peu de sens dans la classe `Piece`. On peut n'appeler cette méthode que sur les objets transtypés en `Roi`.

```
for ( ... )
    dynamic_cast<Roi*>( m_cases[ i ] )->roque();
```

Polymorphisme ; Transtypage v

C++11 fournit 4 opérateurs de transtypage

- `reinterpret_cast` : correspond au transtypage de type C (aucun contrôle n'est affectué entre le type d'origine et le type transtypé)
- `const_cast`: permet d'agir sur la constance de l'objet à transtyper (généralement pour la retirer)
- `static_cast`: permet une vérification statique (à la compilation) de la validité du transtypage
- `dynamic_cast`: permet de s'assurer à l'exécution que le type dans lequel on essaye de transtyper un objet correspond au type réel (dynamique) de l'objet

Méthodes virtuelles pures ; classes abstraites i

- Parfois, on veut spécifier un comportement sans mettre de comportement par défaut (comme une interface en JAVA). On précise alors que la méthode est virtuelle pure et on écrit **= 0** à la fin de la déclaration du prototype.

```
// Piece.h
class Piece { ...
  virtual bool mouvementValide( Echiquier & e, int x, int y ) = 0;
};
```

- Les classes qui dérivent de Piece doivent alors réaliser concrètement ces méthodes, sinon elles ne sont pas instanciables.
- Une classe non instanciable est une classe abstraite.

Méthodes virtuelles pures ; classes abstraites ii

- NB: en C++ on simule les interfaces JAVA en écrivant des classes ne comportant que des méthodes virtuelles pures et un destructeur virtuel.

Vérifiez que `Piece` n'est plus instanciable avec cette méthode virtuelle pure.
Transformez la classe `Joueur` en classe abstraite.

Héritage Multiple i

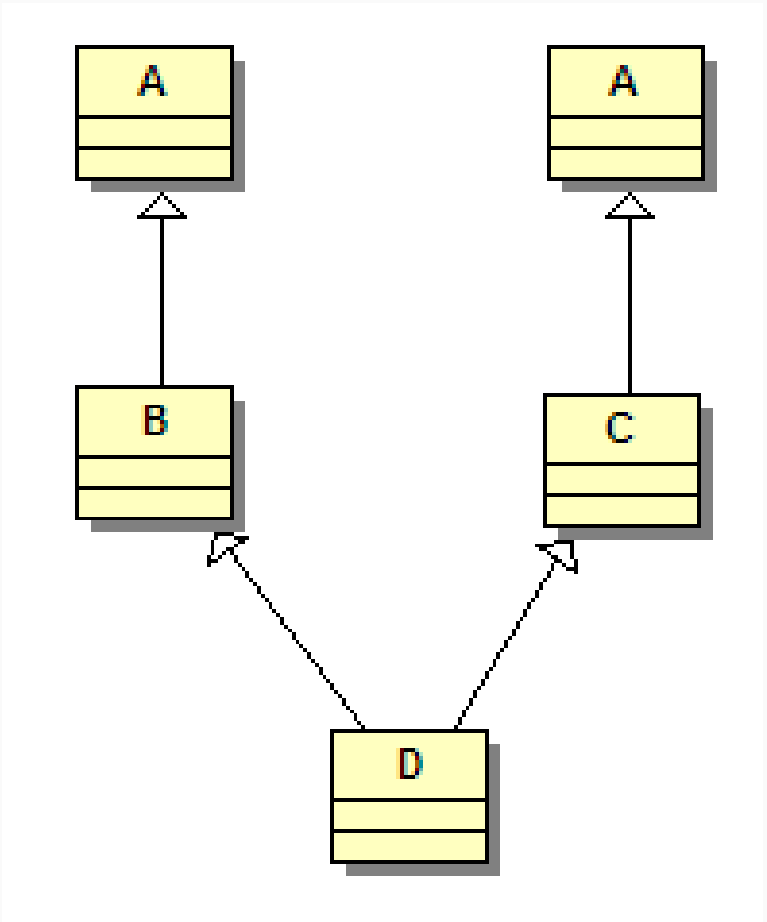
- En C++, il est possible d'hériter de plusieurs classes (mais pas plusieurs fois de la même classe).

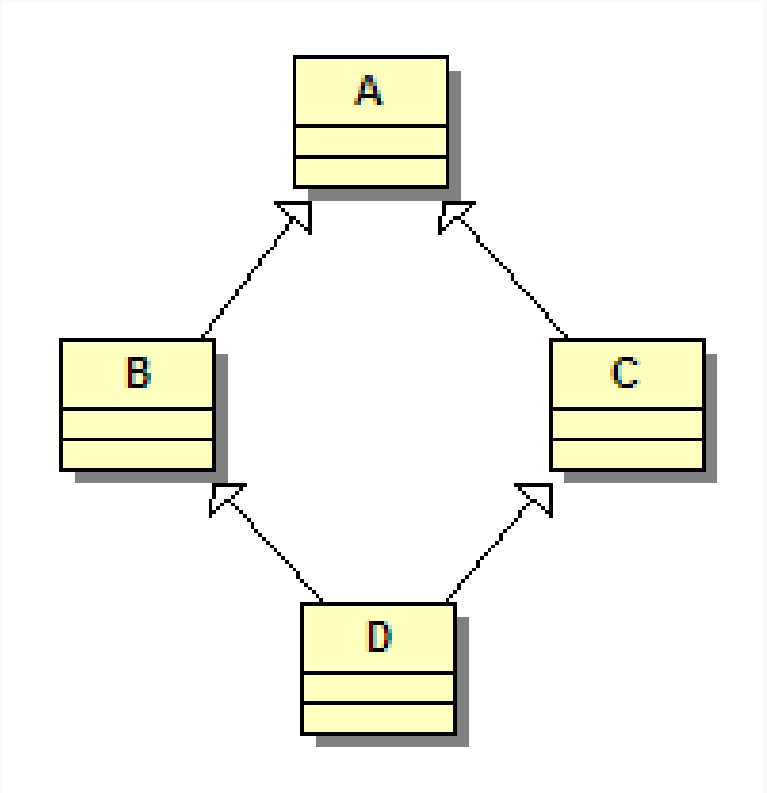
```
// Piece.h
class Reine : public Fou, public Tour { ... };

bool Reine::mouvementValide( Echiquier & e, int x, int y ) {
  return Fou::mouvementValide( e, x, y ) ||
         Tour::mouvementValide( e, x, y );
}
```

- Cependant, pour cet exemple, on souhaite réaliser un héritage en diamant. Mais en l'écrivant de cette manière la classe `Piece` (et ses données membres) sera dupliquée !

Héritage Multiple ii





- Pour obtenir un héritage en diamant, il faut utiliser l'héritage virtuel.

Héritage Multiple iii

```
// Piece.h
class Fou   : virtual public Piece { ... };
class Tour  : virtual public Piece { ... };
class Reine : public Fou, public Tour { ... };
```

- Un problème subsite, comment la partie `Piece` va-t-elle être construite ? Il a été décidé de ne choisir ni la construction par `Fou` ni par `Tour`, mais d'imposer au constructeur de `Reine` de spécifier directement comment la partie `Piece` sera construite.
  Tout appel au constructeur de `Piece` depuis `Fou` ou `Tour` sera ignoré lors de la construction d'une `Reine`.

Héritage Multiple iv

```
Reine::Reine(bool white) : Piece(4,(white?1:8),white),
                           Fou(white), Tour(white) {};
```

Transformez la classe `Reine` avec un héritage multiple. Modifiez la méthode `mouvementValide` pour prendre en compte les déplacements des pieces `Fou` et `Tour`.

Exercice

Terminez le jeu d'échecs. On notera qu'il faudra gérer la zone où un roi est mis en échec. On pourra sans doute étoffer la classe `Echiquier` pour mémoriser les zones où les rois ne peuvent aller. On ignorera le roque et la prise en passant. Un extension possible du jeu est l'initialisation d'un jeu à partir d'un fichier.