

# Programmation C++



**DR. AMINA JARRAYA**

**[JARRAYA.AMINA.FST@GMAIL.COM](mailto:JARRAYA.AMINA.FST@GMAIL.COM)**

# Plan

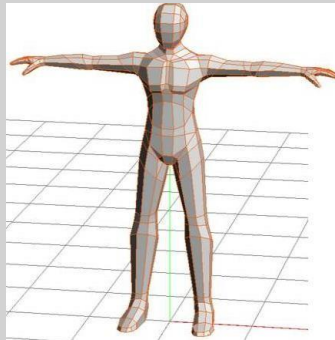


- INTRODUCTION
- DÉFINITION DE L'HÉRITAGE
- TRANSITIVITÉ DE L'HÉRITAGE
- DROITS D'ACCÈS SUR LES MEMBRES HÉRITÉS
- MASQUAGE DANS LA HIÉRARCHIE
- ACCÈS À UNE MÉTHODE MASQUÉE
- HÉRITAGE ET CONSTRUCTEUR
- HÉRITAGE ET DESTRUCTEUR

# Introduction



Commençons par un exemple concret : les personnages de jeux vidéo



# Introduction



## Class Guerrier

string nom  
int energie  
int duree\_vie

Arme arme

Rencontrer(Personnage&)

## Class Voleur

string nom  
int energie  
int duree\_vie

Rencontrer(Personnage&)

Voler(Personnage&)

## Class Magicien

string nom  
int energie  
int duree\_vie

Baguette baguette

Rencontrer(Personnage&)

## Class Sorcier

string nom  
int energie  
int duree\_vie

Baguette baguette

Baton baton

Rencontrer(Personnage&)

## PROBLEMES ?

- ✓ Duplication de codes
- ✓ Problèmes de maintenance : Supposons qu'on veuille changer le nom ou le type d'un attribut, il faudra le faire pour chacune des classes !!!!!



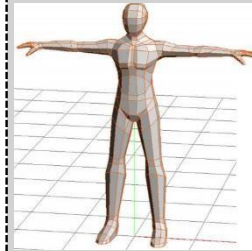
**Solution : Héritage**

# Introduction



## Class Personnage

```
string nom  
int energie  
int duree_vie  
Rencontrer(Personnage&)
```



Pourquoi ne pas regrouper les caractéristiques en commun dans une super classe ?

## Class Voleur

```
Voler(Personnage&)
```



## Class Magicien

```
Baguette baguette
```



## Class Guerrier

```
Arme arme
```



## Class Sorcier

```
Baton baton
```



# Définition de l'héritage



8

# Objets : quatre concepts de base

7

Un des objectifs principaux de la notion **d'objet** :

□ **organiser** des programmes complexes grâce aux notions :

- d'encapsulation

- d'abstraction

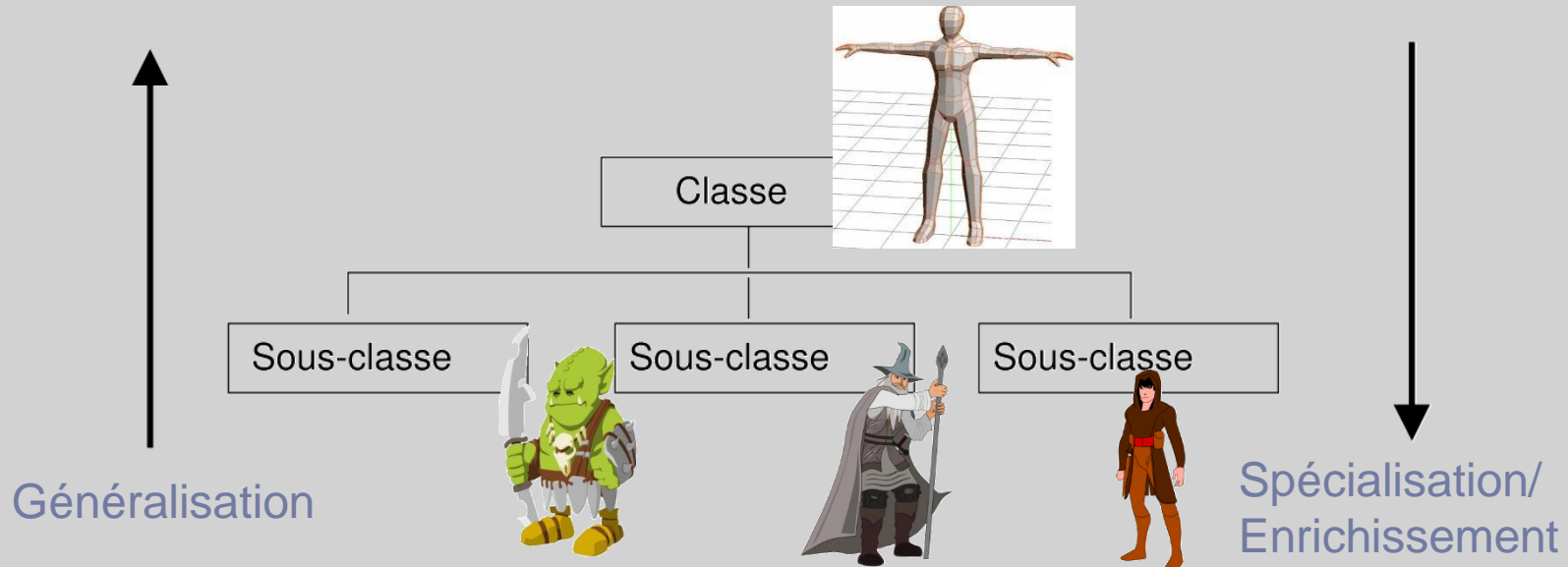
- **d'héritage**

- et de polymorphisme

# Héritage

8

- L'héritage représente la relation «**est-un**».
- Il permet de créer des classes plus **spécialisées**, appelées **sous- classes (ou classes filles)**, à partir de classes plus **générales** déjà existantes, appelées super-classes (**ou classes mères**).





# Héritage en C++

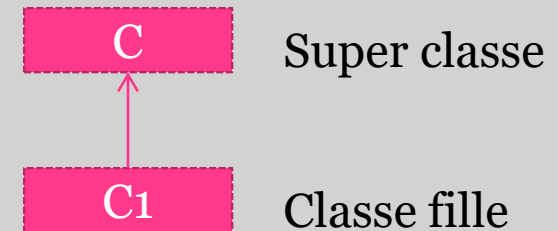
9

- L'héritage permet de donner à une classe toutes les caractéristiques d'une ou de plusieurs autres classes
- Lorsqu'une sous-classe **C1** est créée à partir d'une super-classe **C** :

- le type est hérité : un C1 est (aussi) un C

- C1 hérite l'ensemble :

- □ des attributs de C
- □ des méthodes de C (sauf les constructeurs et destructeur)



➡ Les attributs et les méthodes de C vont être disponibles pour C1 sans que l'on ait besoin de les redéfinir explicitement dans C1

# Héritage en C++

10

Par ailleurs :

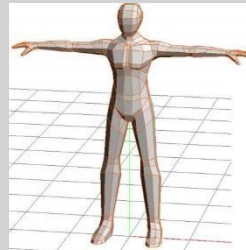
- ❑ des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe C1 : C'est l' enrichissement
- ❑ des méthodes héritées de C peuvent être redéfinies dans C1 : C'est la spécialisation

# Exemple

11

- Lorsqu'une sous-classe **C<sub>1</sub>** (ici Guerrier ou Voleur ) est créée à partir d'une super-classe **C** (ici Personnage),
- le type est hérité : un Guerrier **est aussi** un Personnage :

```
Personnage p;  
Guerrier g;  
// ...  
p = g;  
// ...  
void afficher(Personnage  
const&);  
// ...  
afficher(g);
```



## Question :

Quel partie du guerrier  
est copié dans le  
personnage avec  
l'affectation **p=g;**

## Réponse :

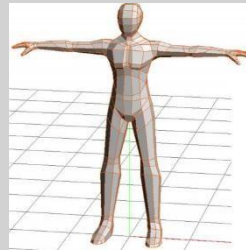
C'est la partie commune  
entre le guerrier et le  
personnage qui est copié  
du guerrier vers le  
personnage

# Exemple

12

- Lorsqu'une sous-classe **C<sub>1</sub>** (ici Guerrier ou Voleur ) est créée à partir d'une super-classe **C** (ici Personnage),
- le type est hérité : un Guerrier **est aussi** un Personnage :

```
Personnage p;  
Guerrier g;  
// ...  
p = g;  
// ...  
void afficher(Personnage  
const&);  
// ...  
afficher(g);
```



Question :

Peut-on inverser l'affectation  
et avoir donc **g=p;**  
Argumentez

Réponse :

**NOOOOOOOOOOOOOOOON**

Vu que les propriétés du  
guerrier ne se trouvent pas  
dans le personnage !!!!

Déjà on ne peut pas dire que  
tout personnage est un guerrier

# Exemple

13

- Lorsqu'une sous-classe **C1** (ici Guerrier) est créée à partir d'une super-classe **C** (ici Personnage) :
  - le Guerrier va hériter de l'ensemble des attributs et des méthodes de Personnage (sauf les constructeurs et destructeur)
  - des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe Guerrier : **arme**
  - des méthodes héritées de Personnage peuvent être redéfinies dans Voleur : **rencontrer(Personnage&)**

# Héritage

14

L'héritage permet donc :

- ❑ d'expliciter des relations structurelles et sémantiques entre classes
- ❑ de réduire les redondances de description et de stockage des propriétés



**L'héritage décrit la relation « est-un »  
et non la relation « a-un »**

**Quel est alors la notion de l'orienté objet qui décrit la  
relation « a-un » ?**

## L'ENCAPSULATION

# Transitivité de l'héritage



17

# Transitivité de l'héritage

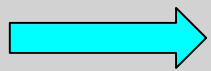
16

Par **transitivité**, les instances d'une sous-classe possèdent :

- les attributs et méthodes (hors constructeurs/destructeur) de l'ensemble des classes parentes (super-classe, super- super-classe, etc.)

## Enrichissement par héritage :

- crée un réseau de dépendances entre classes,
- ce réseau est organisé en une structure arborescente où chacun des nœuds hérite des propriétés de l'ensemble des nœuds du chemin remontant jusqu'à la racine.

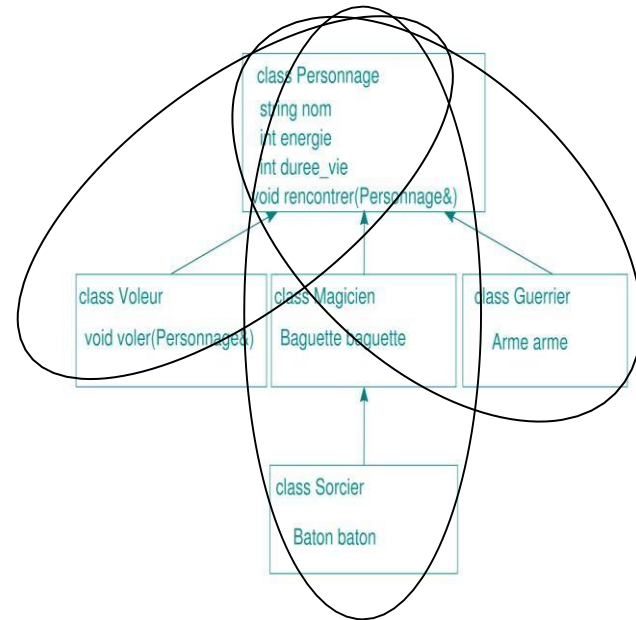
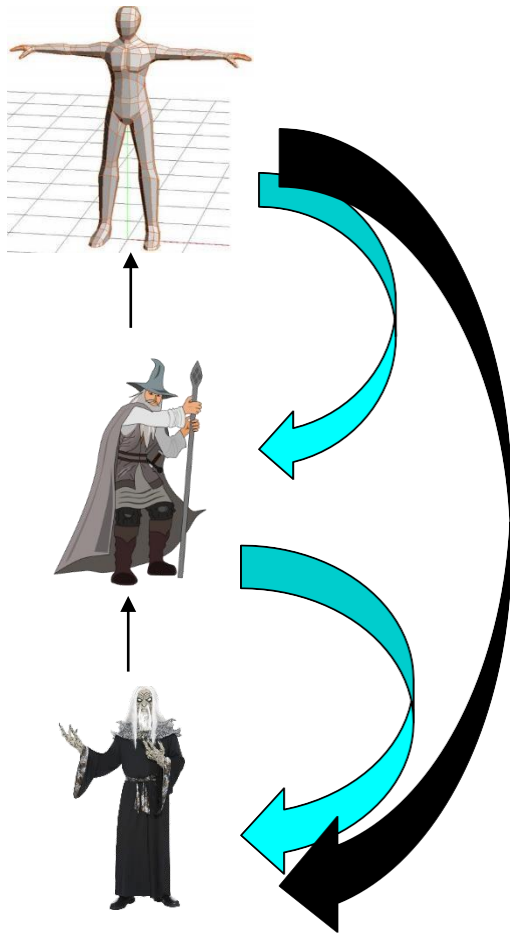


**ce réseau de dépendances définit une hiérarchie de classes**



# Transitivité de l'héritage

17



# L'héritage : résumons ce qu'on a vu

18

## Sous-classe, Super-classes

### □ Une **super-classe** :

- est une classe « **parente** »
- déclare les attributs/méthodes communs
- peut avoir plusieurs sous-classes

### □ Une **sous-classe** est :

- une classe « **enfant** »
  - **étend** une (ou plusieurs) **super-classe(s)**
  - hérite des attributs, des méthodes et du type de la super-classe
- Un attribut/une méthode hérité(e) peut s'utiliser comme si il/elle était déclaré(e) dans la sous-classe au lieu de la super-classe (**en fonction des droits d'accès (plus loin !!)**)
- On **évite** ainsi la **duplication** de **code**

# Syntaxe

19

## □ Comment définir une classe fille C++

```
class NomSousClasse : public NomSuperClasse
{
    // Déclaration des attributs et méthodes spécifiques à la sous-classe
};
```

```
class Guerrier : public Personnage
{
    //...
    private:
        Arme arme;
};
```

```
class Rectangle : public FigureGeometrique
{
    //...
    private:
        double largeur; double hauteur;
};
```

# Droits d'accès



22


# Droits d'accès sur les membres hérités

21

## Question :

- Est-ce qu'une classe fille peut accéder à tous les attributs et méthodes de la classe mère qu'ils soient publics ou privés

Si les attributs sont public, quel problème peut se poser ?



Même si je n'hérite pas de la classe mère, je peux briser l'interface et accéder à tous ses attributs, comme ça j'aurai un accès total à tous les attributs et méthodes.

Si private?

Je suis la classe fille et j'ai réellement besoin d'accéder aux attributs et méthodes privés, que faire ?



# Droits d'accès sur les membres hérités

22

Rappel : Jusqu'à maintenant, l'accès aux membres (attributs et méthodes) d'une classe pouvait être :

- soit public : visibilité totale à l'intérieur et à l'extérieur de la classe (mot-clé **public** )
- soit privé : visibilité uniquement à l'intérieur de la classe (mot-clé **private** )

Un troisième type d'accès régit l'accès aux attributs/méthodes au sein d'une hiérarchie de classes :

- l'accès protégé : assure la visibilité des membres d'une classe dans les classes de sa descendance
- Le mot clé est « **protected** ».

# Accès protégé

23

- ❑ Le niveau d'accès protégé correspond à une extension du niveau privé permettant l'accès aux sous-classes.

```
class Personnage {  
    // ...  
protected:  
    int energie;  
};  
class Guerrier : public Personnage {  
public:  
    // ...  
    void frapper(Personnage& le_pauvre) {  
        if (energie > 0) {  
            // frapper le perso  
        }  
    }  
};
```

Que se passera t-il si on changeait protected par private?

# Accès protégé : portée

24

- Le niveau d'accès protégé correspond à une extension du niveau privé permettant l'accès aux sous-classes... **mais uniquement dans leur portée (de sous-classe)**, et non pas dans la portée de la super-classe

```
class A {
//...
protected:  int a;
private:    int prive;
};

class B: public A {
public:
    //...
    void f(B autreB, A autreA, int x) {
        a = x;  // OK A::a est protected => acces possible
        // prive = x; // erreur : A::prive est private

        // a += autreB.prive; // erreur (meme raison)
        a += autreB.a; // OK : dans la meme classe (B)
        // a += autreA.a; // INTERDIT ! : this n'est pas de la meme
                           // classe que autreA (role externe)
    }
};
```



# Accès protégé : portée

25

- Les niveaux d'accès peuvent être modifiés lors de l'héritage

## Syntaxe :

```
class ClasseEnfant: [accès] classeParente
{
    /* Déclaration des membres
       spécifiques à la sous-classe */
    //...
};
```

- où **accès** est le mot-clé **public**, **protected** ou **private**. Les crochets entourant un élément **[ ]** indiquent qu'il est optionnel.
- Les droits peuvent être conservés ou restreints, **mais jamais relâchés !**
- Par défaut, l'accès est privé.

# Droits d'accès sur les membres hérités (récap)

26

- ≡ Membres **publics** : accessibles pour les **programmeurs et utilisateurs** de la classe
- ≡ Membres **protégés** : accessibles aux **programmeurs d'extensions** par héritage de la classe (**visible dans la sous classe mais pas pour les utilisateurs des sous classes**)
- ≡ Membres **privés** : pour le **programmeur de la classe** : structure interne

# Masquage



28

# Masquage dans la hiérarchie

28

Supposons qu'on ait le scénario suivant : A la rencontre d'un personnage tous les personnages saluent le personnage sauf le guerrier qui frappe tous ce qu'il voit

- Pour un personnage non- Guerrier :  

```
void rencontrer(Personnage& le_perso) const {  
    saluer(le_perso);  
}
```
- Pour un Guerrier  

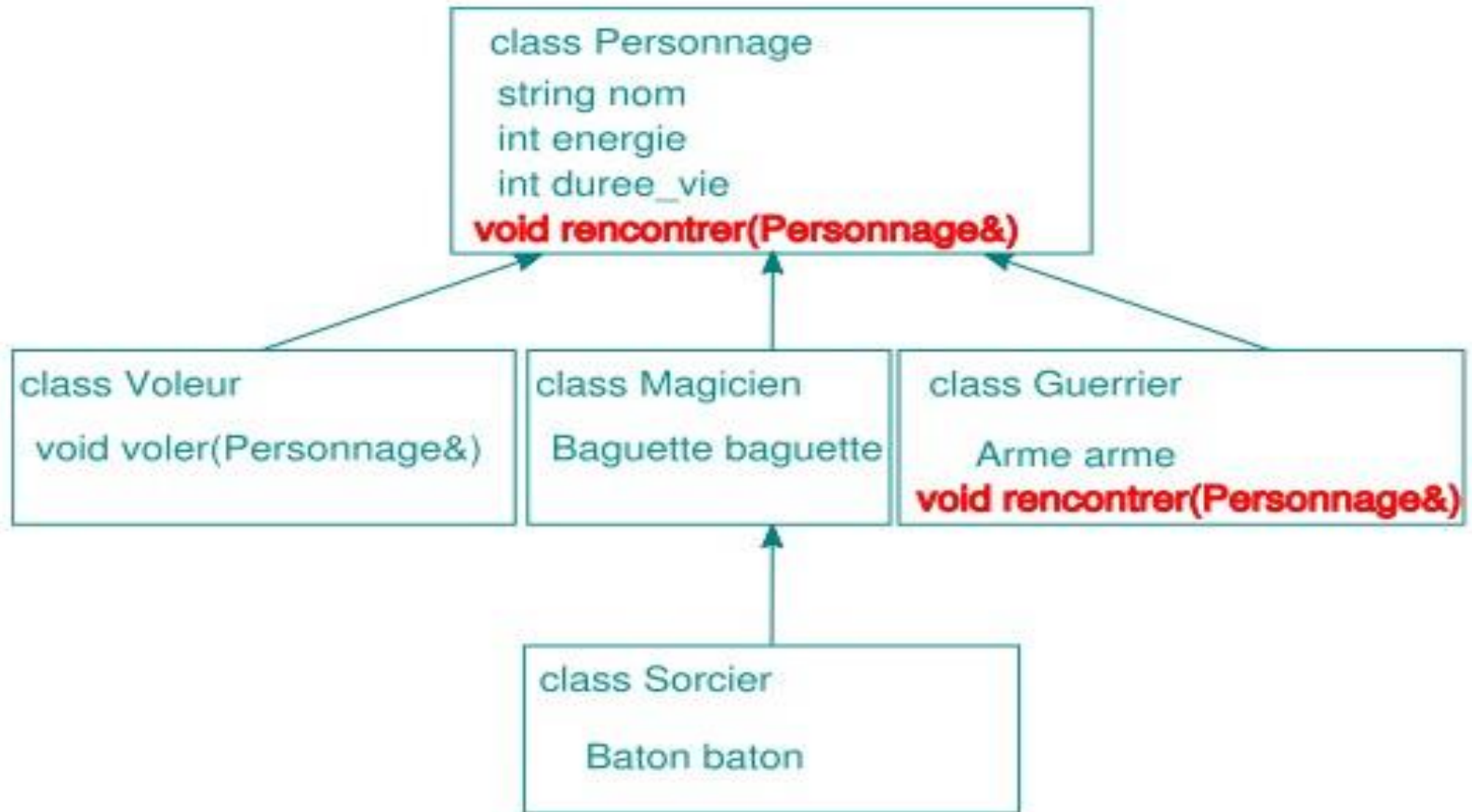
```
void rencontrer(Personnage& le_pauvre) const {  
    frapper(le_pauvre);  
}
```
- Faut-il re-concevoir toute la hiérarchie ?



 **Non, on ajoute simplement une méthode rencontrer(Personnage&) spéciale dans la sous-classe Guerrier**

# Masquage dans la hiérarchie

29



# Masquage dans la hiérarchie

30

- ❑ Donc finalement le **masquage** est : **identificateur qui en cache un autre**
- ❑ Situations possibles dans une hiérarchie :
  - Même nom d'attribut ou de méthode utilisé sur plusieurs niveaux
  - **Peu courant pour les attributs**
  - **Très courant et pratique pour les méthodes**

# Accès à une méthode masquée

31

- ❑ Supposons que lorsqu'on va masquer une méthode, on ait besoin de lui accéder que faire ?!!

Prenant l'exemple suivant :

- ❑ Même toujours énervé, le Guerrier reste toujours poli et commence par saluer le personnage rencontré avant de le frapper.

Doit-on reprendre tous le code de la méthode masquée?

- ❑ Le nouveau fonctionnement

1. Personnage non- Guerrier :

- Méthode générale (rencontrer de Personnage)

2. Personnage Guerrier :

- Méthode spécialisée (rencontrer de Guerrier)
- Appel à la méthode générale depuis la méthode spécialisée

# Accès à une méthode masquée

32

**D'après vous quelle solution peut nous permettre d'utiliser la méthode masquée?**

**Astuce** : pensez à la façon avec la quelle on définit une méthode à l'extérieur de la classe et comment on accède aux attributs et méthodes static.

Pour accéder aux attributs/méthodes masqué(e)s

- on utilise **l'opérateur de résolution de portée ::**
- Syntaxe : **NomClasse::méthode ou attribut**



```
class Guerrier : public Personnage {  
    //...  
    void rencontrer (Personnage& perso) {  
        Personnage::rencontrer(perso); // salutation d'usage !!  
        frapper(perso);  
    }  
};
```



# Héritage et constructeurs



34

# Héritage et constructeur

34

- ❑ Question : Que se passe t-il lors de l'instanciation d'un objet?
- ❑ Le constructeur est appelé et généralement, il construit notre objet en le préparant et en initialisant ces attributs.
- ❑ Question : Est-ce que c'est toujours le cas pour l'héritage sachant que la classe fille hérite des propriétés de la classe mère? Si oui qui fait quoi  
!!!!

# Héritage et constructeur

35

- Lors de l'instanciation d'une sous-classe, il faut initialiser
  - les attributs propres à la sous-classe
  - les attributs hérités des super-classes
- ...il ne doit pas être à la charge du concepteur des sous-classes de réaliser lui-même l'initialisation des attributs hérités.. **Comment faire alors?!!!!**
- **Indice** : pensez à la solution que nous avons trouvée pour instancier des attributs objet d'une classe.



**Solution** : l'initialisation des attributs hérités doit se faire en invoquant les constructeurs des super-classes.

# Héritage et constructeur

36

- L'invocation du constructeur de la super-classe se fait au début de la section d'appel aux constructeurs des attributs.
- Syntaxe :

```
SousClasse(liste de paramètres) : SuperClasse(Arguments),  
attribut1(valeur1), ..., attributN(valeurN)  
{  
    // corps du constructeur  
}
```

- Lorsque la super-classe admet un constructeur par défaut, l'invocation explicite de ce constructeur dans la sous-classe n'est pas obligatoire
- le compilateur se charge de réaliser l'invocation du constructeur par défaut

# Héritage et constructeur

37

- Si la classe parente n'admet pas de constructeur par défaut, l'invocation explicite d'un de ses constructeurs est obligatoire dans les constructeurs de la sous-classe
- La sous-classe doit admettre au moins un constructeur explicite.

```
class FigureGeometrique {  
protected:    Position position;  
public:  
    FigureGeometrique(double x, double y) : position(x, y) {}  
    // ...  
};  
  
class Rectangle : public FigureGeometrique {  
protected:    double largeur; double hauteur;  
public:  
    Rectangle(double x, double y, double l, double h)  
        : FigureGeometrique(x,y), largeur(l), hauteur(h) {}  
    // ...  
};
```


# Héritage et constructeur

38

- Autre exemple (avec constructeur par défaut)

```
class FigureGeometrique {
protected:    Position position;
public:
    /* Note : le constructeur par défaut par défaut de FigureGeometrique
     *        appelle le constructeur par défaut de Position.
     */
    // ...
};

class Rectangle : public FigureGeometrique {
protected:    double largeur; double hauteur;
public:
    Rectangle(double l, double h)
        : largeur(l), hauteur(h)
    {}
    // ...
};
```



Appel implicite du constructeur par défaut de FigureGeometrique

# Héritage et constructeur

39

- Autre exemple: Parfois il n'est pas nécessaire d'avoir des attributs supplémentaires...

```
class Carre : public Rectangle {  
public:  
    Carre(double taille)  
    : Rectangle(taille, taille)  
    {}  
    /* Et c'est tout !  
       (sauf s'il y avait des manipulateurs,  
       il faudrait alors sûrement aussi les  
       redéfinir)  
    */  
};
```

# Héritage et constructeur: Résumé

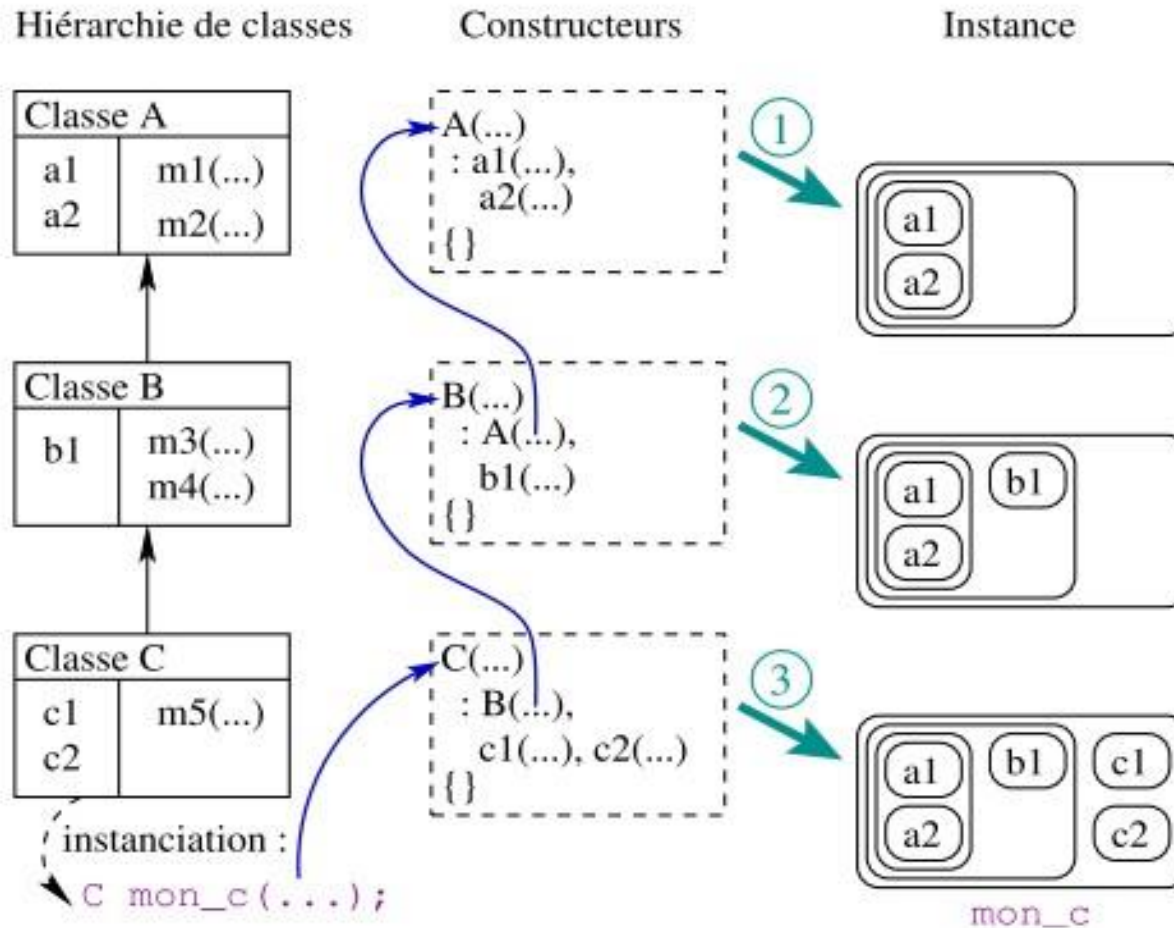
41

- ❑ Chaque constructeur d'une sous-classe doit appeler un des constructeurs de la super-classe,
- ❑ L'appel est trouvée en premier dans la liste d'initialisation
- ❑ Et si on oublie l'appel à un constructeur de la super-classe?
  - Appel automatique au constructeur par défaut de la super classe.
  - Pratique parfois, mais **ERREUR** si le constructeur par défaut n'existe pas



# Ordre d'appel des constructeurs

41



# Ordre d'appel des destructeurs

42

- ❑ Les destructeurs sont toujours appelés dans l'ordre inverse (/symétrique) des constructeurs.
- Par exemple dans l'exemple précédent, lors de la destruction d'un C , on aura appel et exécution de :
  - C::~~C()
  - B::~~B()
  - A::~~A()
- ❑ et dans cet ordre puisque les constructeurs avaient été appelés dans l'ordre
  - A::A()
  - B::B()
  - C::C()

# Héritage des constructeurs avec



43

- ❑ Les constructeurs ne sont pas hérités en général, mais en, **C++11**, on peut demander leur héritage en utilisant le mot clé « using ».
- ❑ On récupère alors **tous** les constructeurs de la super-classe, càd on peut construire la sous classe avec les mêmes arguments, **mais, attention!** Ces constructeurs n'initialisent donc pas les attributs spécifiques de la sous-classe.
- ❑ Plus utilisé lorsqu'on n'as pas de nouvel attribut dans la sous-classe.

```
class A {  
public:  
    A(int);  
    A(double, double);  
    // ...  
};
```

```
class B : public A {  
using A::A;  
/* existent alors maintenant  
    B::B(int)  
    et B::B(double, double) */  
};
```

# Quiz



- Sortie ?
  - Inside P
  - Inside Q
  - Erreur de compilation

```
#include<iostream>

using namespace std;
class P {
public:
    void print() { cout <<" Inside P"; }
};

class Q : public P {
public:
    void print() { cout <<" Inside Q"; }
};

class R: public Q { };

int main(void)
{
    R r;
    r.print();
    return 0;
}
```

# Quiz



- Sortie ?

- Inside P

- **Inside Q**

- Erreur de compilation

La fonction print() n'est pas présente dans la classe R. Elle est donc recherchée dans la hiérarchie d'héritage. print () est présente dans les deux classes P et Q, laquelle devrait être appelée? L'idée est que, s'il existe un héritage à plusieurs niveaux, la fonction est recherchée linéairement dans la hiérarchie d'héritage jusqu'à ce que print() soit trouvée.

```
#include<iostream>

using namespace std;
class P {
public:
    void print() { cout <<" Inside P"; }
};

class Q : public P {
public:
    void print() { cout <<" Inside Q"; }
};

class R: public Q { };

int main(void)
{
    R r;
    r.print();
    return 0;
}
```

# Quiz



- Sortie ?

- $i = 0$   $j = 0$
- Erreur de compilation :  $i$  et  $j$  sont privées
- Erreur de compilation : impossible d'appeler le constructeur de Base

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

class Base {
private:
    int i, j;
public:
    Base(int _i = 0, int _j = 0): i(_i), j(_j) { }
};

class Derived: public Base {
public:
    void show(){
        cout<<" i = "<<i<<" j = "<<j;
    }
};

int main(void) {
    Derived d;
    d.show();
    return 0;
}
```

# Quiz



- Sortie ?
  - $i = 0$   $j = 0$
  - **Erreur de compilation : i et j sont privées**
  - Erreur de compilation : impossible d'appeler le constructeur de Base

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

class Base {
private:
    int i, j;
public:
    Base(int _i = 0, int _j = 0): i(_i), j(_j) { }
};

class Derived: public Base {
public:
    void show(){
        cout<<" i = "<<i<<" j = "<<j;
    }
};

int main(void) {
    Derived d;
    d.show();
    return 0;
}
```

Pour que i et j soient accessibles depuis la classe Derived, il faut changer la portée soit public soit protected.

# Quiz



- Sortie ?
  - Pas d'erreur de compilation
  - Erreur de compilation dans la ligne "Base \*bp = new Derived;"
  - Erreur de compilation dans la ligne "Derived \*dp = new Base;"

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

class Base {};

class Derived: public Base {};

int main()
{
    Base *bp = new Derived;
    Derived *dp = new Base;
}
```



# Quiz



- Sortie ?
  - Pas d'erreur de compilation
  - Erreur de compilation dans la ligne "Base \*bp = new Derived;"
  - **Erreur de compilation dans la ligne "Derived \*dp = new Base;"**

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

class Base {};

class Derived: public Base {};

int main()
{
    Base *bp = new Derived;
    Derived *dp = new Base;
}
```

Base \*bp = new Derived; → OK // Derived est une Base  
Derived \*dp = new Base; → KO // Base n'est pas une Derived

# Quiz



- Sortie ?
  - Erreur de compilation dans la ligne "bp->show()"
  - Erreur de compilation dans la ligne "cout << bp->x"
  - In Base 10
  - In Derived 10

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

class Base
{
public:
    void show()
    {
        cout<<" In Base ";
    }
};

class Derived: public Base
{
public:
    int x;
    void show()
    {
        cout<<"In Derived ";
    }
    Derived()
    {
        x = 10;
    }
};

int main(void)
{
    Base *bp, b;
    Derived d;
    bp = &d;
    bp->show();
    cout << bp->x;
    return 0;
}
```

# Quiz



- Sortie ?
  - Erreur de compilation dans la ligne " bp->show()“
  - **Erreur de compilation dans la ligne "cout << bp->x”**
  - In Base 10
  - In Derived 10

Le pointeur de classe de base bp peut pointer vers un objet de classe dérivé, mais nous ne pouvons accéder qu'aux membres (attributs et méthodes) de la classe de base en utilisant le pointeur bp.

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

class Base
{
public:
    void show()
    {
        cout<<" In Base ";
    }
};

class Derived: public Base
{
public:
    int x;
    void show()
    {
        cout<<"In Derived ";
    }
    Derived()
    {
        x = 10;
    }
};

int main(void)
{
    Base *bp, b;
    Derived d;
    bp = &d;
    bp->show();
    cout << bp->x;
    return 0;
}
```

# Quiz



- Sortie ?
  - Base::fun(int i) appelé
  - Derived::fun() appelé
  - Base::fun() appelé
  - Erreur de compilation

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

class Base
{
public:
    int fun() { cout << "Base::fun() called"; }
    int fun(int i) { cout << "Base::fun(int i) called"; }
};

class Derived: public Base
{
public:
    int fun() { cout << "Derived::fun() called"; }
};

int main()
{
    Derived d;
    d.fun(5);
    return 0;
}
```

# Quiz



- Sortie ?
  - Base::fun(int i) appelé
  - Derived::fun() appelé
  - Base::fun() appelé
  - **Erreur de compilation**

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

class Base
{
public:
    int fun() { cout << "Base::fun() called"; }
    int fun(int i) { cout << "Base::fun(int i) called"; }
};

class Derived: public Base
{
public:
    int fun() { cout << "Derived::fun() called"; }
};

int main()
{
    Derived d;
    d.fun(5);
    return 0;
}
```

Si une classe dérivée écrit sa propre méthode, toutes les fonctions de la classe de base portant le même nom deviennent masquées, même si les signatures des fonctions de la classe de base sont différentes. Dans la question ci-dessus, lorsque fun () est réécrite dans Derived, il masque à la fois fun () et fun (int) de la classe de base.

# Quiz



- Sortie ?
  - Base::fun(int i) appelé
  - Erreur de compilation

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

class Base {
public:
    int fun()          {    cout << "Base::fun() called";    }
    int fun(int i)     {    cout << "Base::fun(int i) called"; }
};

class Derived: public Base {
public:
    int fun() {    cout << "Derived::fun() called";    }
};

int main() {
    Derived d;
    d.Base::fun(5);
    return 0;
}
```

# Quiz



- Sortie ?
  - **Base::fun(int i)**  
**appelé**
  - Erreur de compilation

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

class Base {
public:
    int fun()          { cout << "Base::fun() called"; }
    int fun(int i)     { cout << "Base::fun(int i) called"; }
};

class Derived: public Base {
public:
    int fun() { cout << "Derived::fun() called"; }
};

int main() {
    Derived d;
    d.Base::fun(5);
    return 0;
}
```

Nous pouvons accéder aux fonctions de classe de base à l'aide de l'opérateur de résolution de portée :: , même si elles sont masquées par une fonction de classe dérivée.