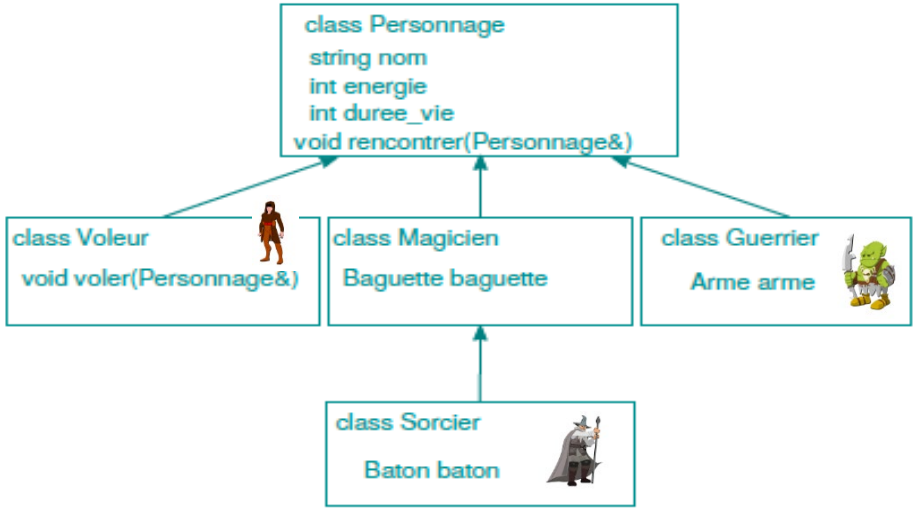
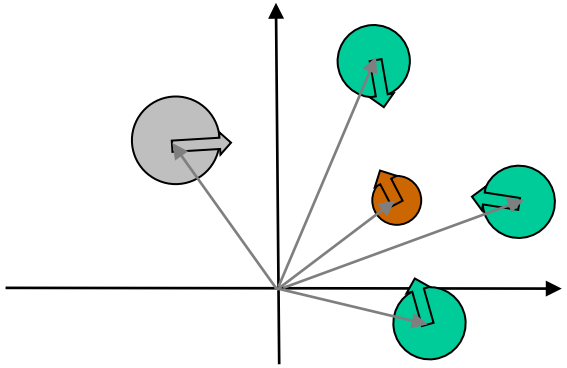
Héritage : Distinguer **“Est un(e)”** de **“Possède un(e)”**

La relation « Possède un(e) » est la relation déjà connue entre une instance d’une classe et ses attributs :
=> Un Personnage possède : nom, energie, duree_vie



La relation « Est un(e) » est celle qui existe entre une sous-classe et sa classe parente:
⇒ **Un Sorcier est un Magicien**
⇒ **Un Magicien est un Personnage**

Question SpeakUp: supposons que pour les besoins d’une simulation de scénario de film, je veux pouvoir éviter des collisions et calculer des stratégies de déplacement. Pour cela l’information de **position**, **orientation** et de **rayon d’espace occupé** des personnages est nécessaire.

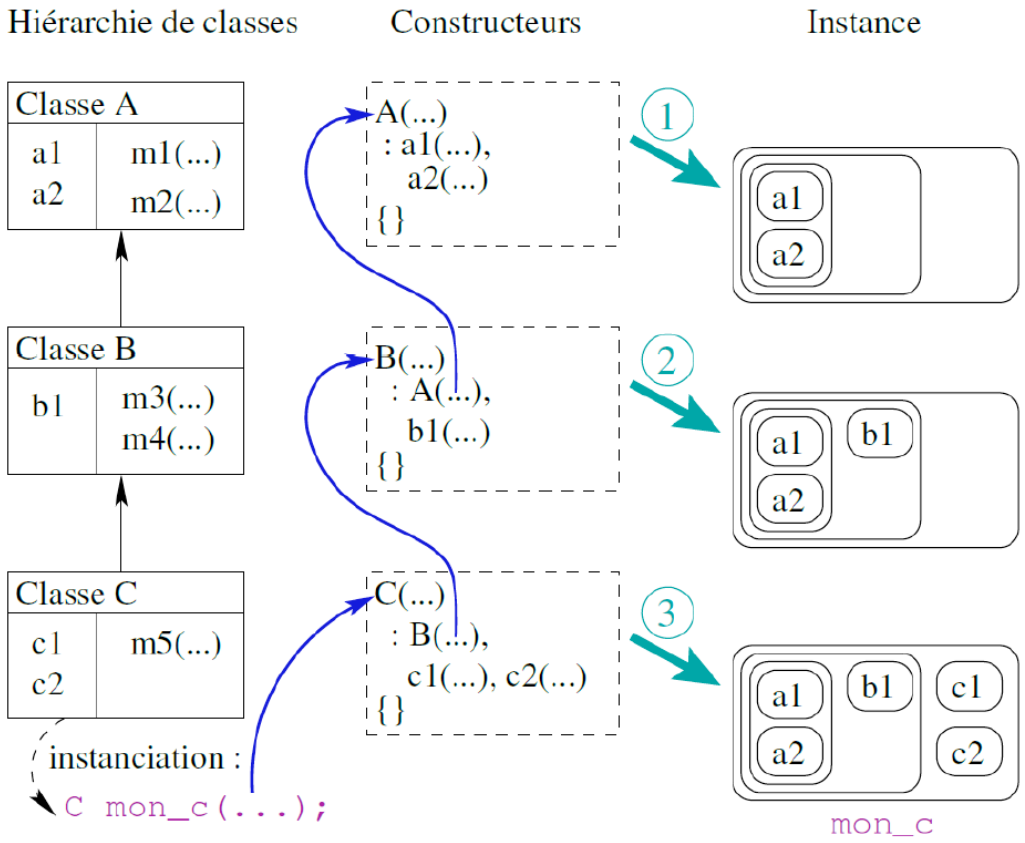


Quelle approche est la plus pertinente :

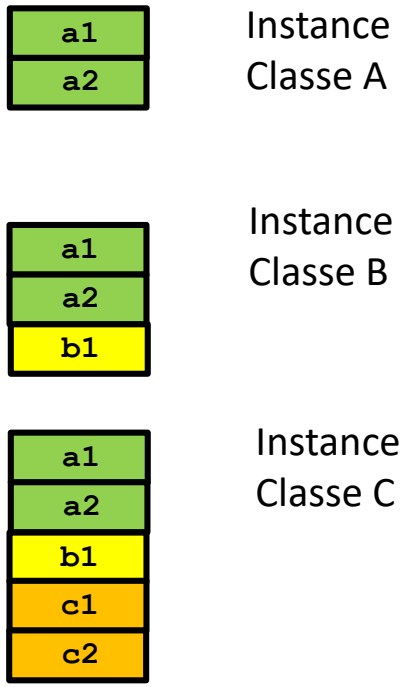
A Créer des nouvelles sous-classes
B Ajouter l’information à chaque sous-classe existante
C Créer une nouvelle superclasse dont Personnage va hériter
D Ajouter l’information à la classe Personnage

Héritage: implémentation d'une instance (BOOC Leçon 19 p43)

Localisation des attributs de la hiérarchie de classes (Fig 1)



En mémoire



Conséquence (slicing)

L'affectation d'une instance d'une classe dérivée à une instance de sa classe parente garde seulement les attributs de la classe parente.

La «tranche» (*slice*) des attributs de la classe dérivée est perdue

```
A x;
B y;
C z;
// autorisé mais perte
x = y;
y = z;
x = z ;
```

Exemple: spécialisation d'une méthode code source rob.cc (1)

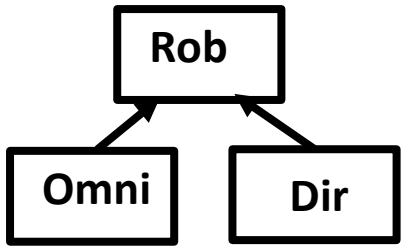
La superclasse **Rob** définit un robot avec un attribut de position **pos** et une méthode **move_to** pour atteindre un but

```
class Rob
{
public:
    Rob(T2D pos={0.,0.}){}
    void move_to(T2D goal) {pos=goal;} // téléportation
    void affiche() {cout << pos[X] << " " << pos[Y] << endl;}
protected:
    T2D pos;
};
```

avec

```
typedef array<double,2> T2D;
enum Coord {X, Y};
```

Deux classe dérivées **Omni** et **Dir** spécialisent la méthode **move_to** pour atteindre un but

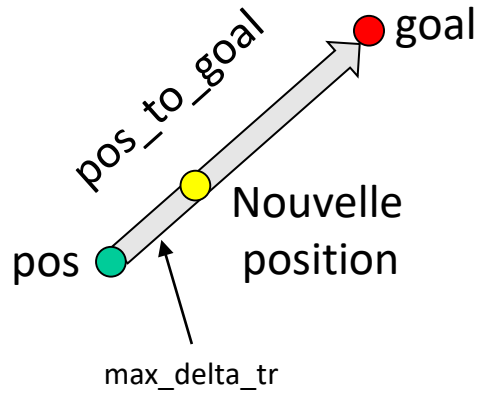


Pour la classe **Omni** la méthode **move_to** déplace l'instance **Omni** en direction du point **goal** mais au plus d'une distance **max_delta_tr**

```
class Omni: public Rob // robot Omnidirectionnel, pas besoin de tourner
{
public:
    Omni ():Rob(){}
    void move_to(T2D goal); // avec limitation en translation
};

// se déplace au plus de max_delta_tr vers le point goal
void Omni::move_to(T2D goal)
{
    T2D pos_to_goal = {goal[X] - pos[X], goal[Y] - pos[Y]} ;
    double norm(t2d_norm(pos_to_goal));

    if(norm <= max_delta_tr) pos = goal;
    else add_scaled_vector(pos, pos_to_goal, max_delta_tr/norm);
}
```



Exemple: spécialisation d'une méthode code source rob.cc (2)

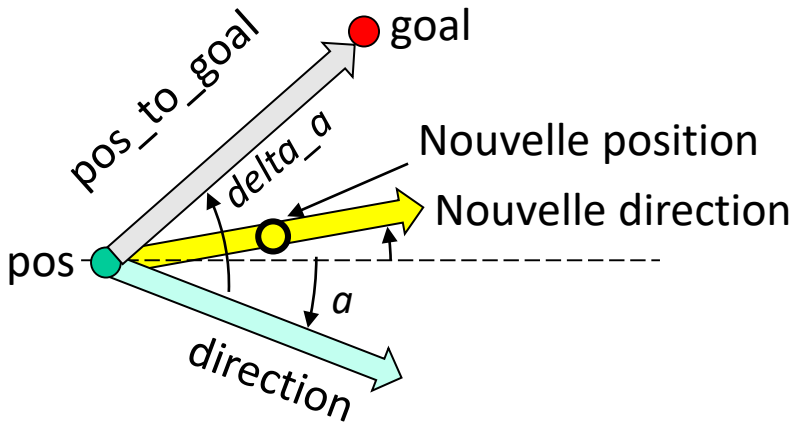
```
class Dir: public Rob // robot directionnel, doit tourner
){
public:
    Dir():Rob(),a(0){}
    void move_to(T2D goal); // tourne puis avance (avec limitations)
private:
    Orient a; // angle représentation l'orientation en rd
};

// s'oriente au plus de max_delta_rt en direction de pos_to_goal
// puis se déplace au plus de max_delta_tr vers le point goal
void Dir::move_to(T2D goal)
){
    T2D pos_to_goal = {goal[X] - pos[X], goal[Y] - pos[Y]};
    double norm (t2d_norm(pos_to_goal));

    Orient goal_a(atan2(pos_to_goal[Y],pos_to_goal[X]));
    Orient delta_a( goal_a - a);

    if(abs(delta_a) <= max_delta_rt)
    {
        a = goal_a ;
        if(norm <= max_delta_tr) pos = goal;
        else add_scaled_vector(pos, pos_to_goal, max_delta_tr/norm);
    }
    else
    {
        a += ((delta_a > 0) ?  1. : -1.)*max_delta_rt ;
        T2D travel_dir = {cos(a), sin(a)}; // direction normalisée
        add_scaled_vector(pos, travel_dir, max_delta_tr);
    }
}
```

Pour la classe **Dir** la méthode **move_to**
oriente l'instance **Omni** en direction du
vecteur **pos_to_goal**
mais au plus de **max_delta_rt**
puis la déplace en direction du point **goal**
mais au plus d'une distance **max_delta_tr**



prog.cc

```
#include <iostream>
class A {
protected:
        int x;
};

class B: public A {
protected:
        int x1;
};

class C: public B {
public:
        void h(A *p1, B* p2, C* p3);
private:
        int x2;
};

void C::h(A *p1, B* p2, C* p3){

        C *p4(new C);

        std::cin >> p1->x ;    // 1
        std::cin >> p2->x1 ;   // 2
        std::cin >> p3->x2 ;   // 3
        std::cin >> p4->x ;    // 4
}
```

Question SpeakUp 21158:

Question Hiérachie de classes (MOOC semaine 4):
On compile (option –c) le fichier source prog.cc déclarant la hiérarchie de classes A-B-C.

On s'intéresse aux 4 lignes de lecture sur std::cin avec des commentaires numérotés de 1 à 4.

Chacune de ces lignes compile-t-elle correctement ?
Réponses proposées (oui/non) dans l'ordre des 4 lignes

	// 1	// 2	// 3	// 4
A	oui	oui	oui	oui
B	non	oui	oui	non
C	non	non	oui	non
D	non	non	oui	oui