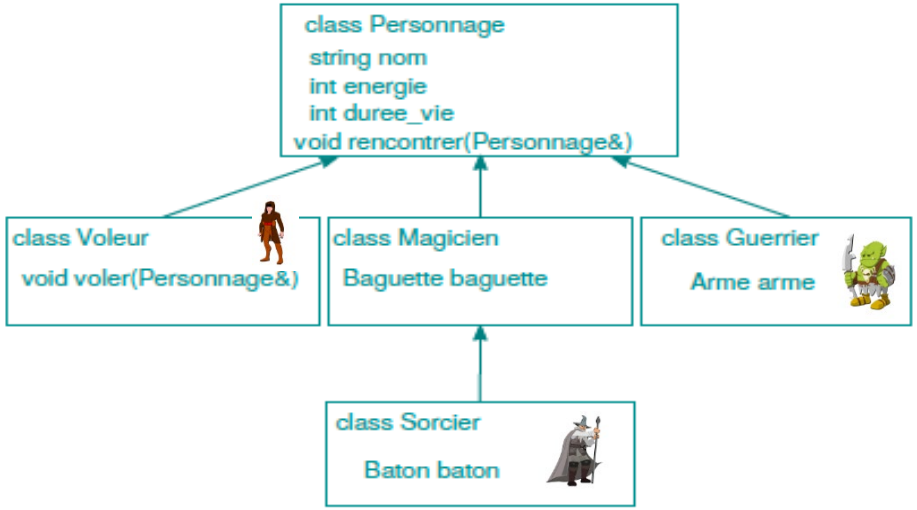
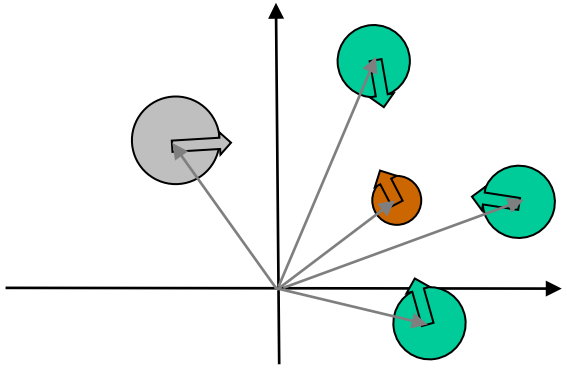
Héritage : Distinguer **“Est un(e)”** de **“Possède un(e)”**

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**

Quizz1: 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.

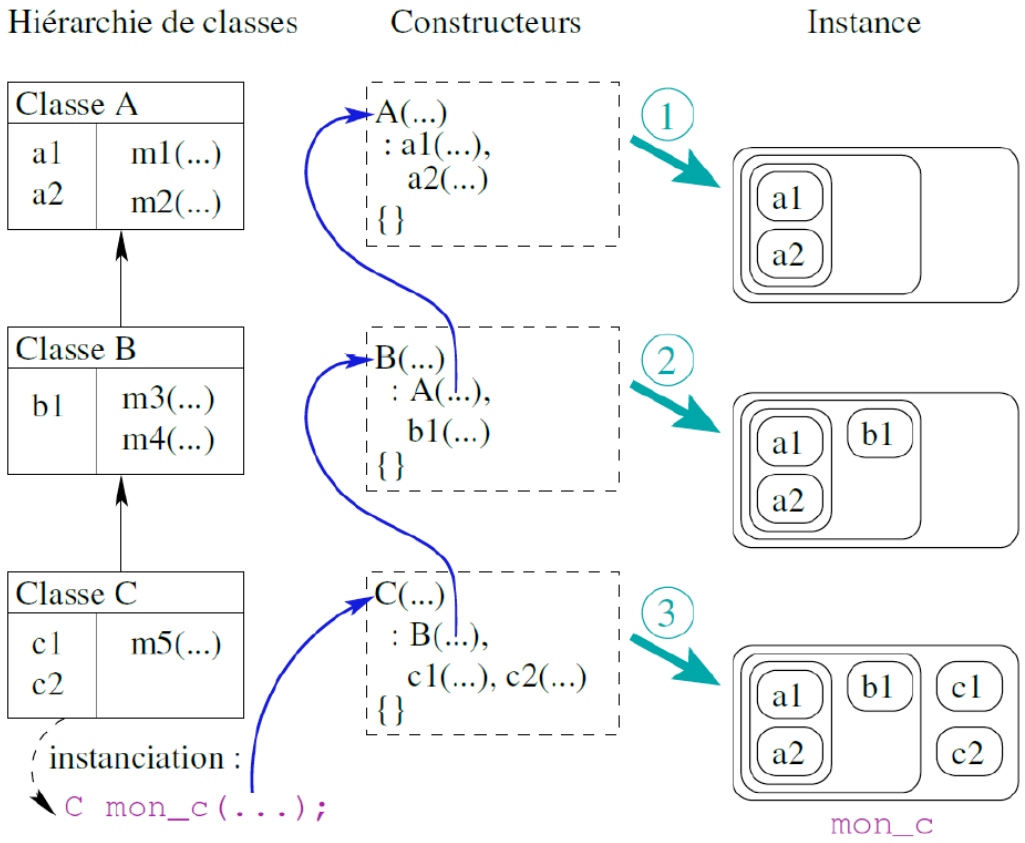


Quelle approche est la plus pertinente :

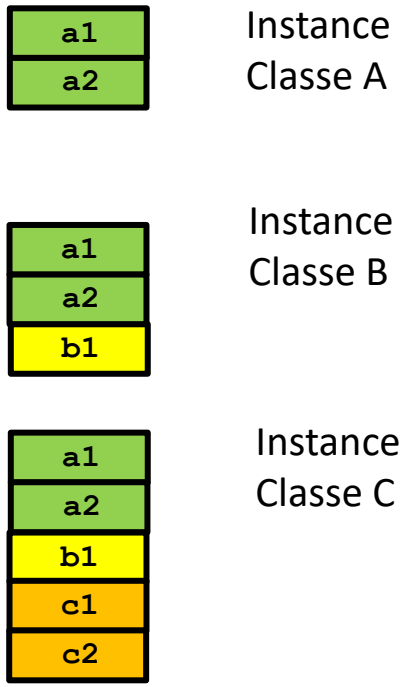
A Créer des nouvelles sous-classes
B Ajouter l’information à chaque sous-classe existante
C Créer une nouvelle superclasse dont Personnage va hériter
D Ajouter l’information à la classe Personnage

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)

On veut définir une hiérarchie de classes de robots ayant différentes méthodes de déplacement dans le plan. Tout d'abord, pour clarifier les manipulations sur les positions et les vecteurs dans le plan 2D on définit un type **S2d** et quelques fonctions utilitaires :

```
typedef array<double,2> S2d;
```

```
enum Coord {X, Y};
```

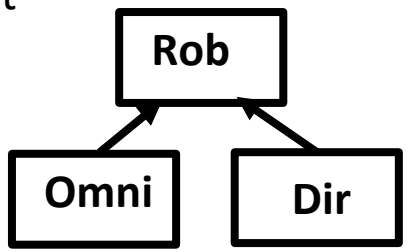
```
double s2d_norm(S2d tab);
double s2d_prod_scal(S2d v1, S2d v2);
void   s2d_add_scaled_vector(S2d& pos, const S2d& pos_to_goal, double scaling);
```

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(S2d pos={0.,0.}):pos(pos){}
    void move_to(S2d goal) {pos=goal;} // téléportation
    void affiche() {cout << pos[X] << "\t" << pos[Y] << endl;}
protected:
    S2d pos;
};
```

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

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

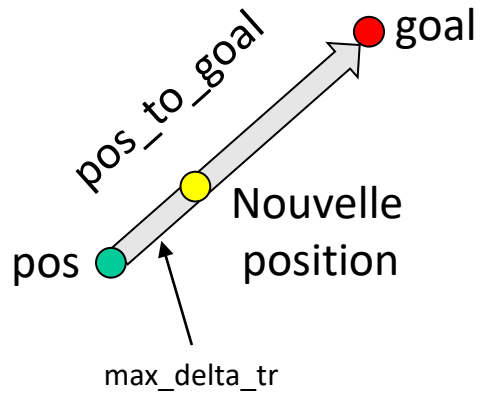



```
class Omni: public Rob // robot Omnidirectionnel, pas besoin de tourner
{
public:
    Omni ():Rob(){}
    void move_to(S2d goal); // avec limitation en translation
};
```

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

```
constexpr double max_delta_tr(5.);
```

```
// se déplace au plus de max_delta_tr vers le point goal
void Omni::move_to(S2d goal)
{
    S2d pos_to_goal = {goal[X] - pos[X], goal[Y] - pos[Y]} ;
    double norm(s2d_norm(pos_to_goal));

    if(norm <= max_delta_tr) pos = goal;
    else s2d_add_scaled_vector(pos, pos_to_goal, max_delta_tr/norm);
}
```

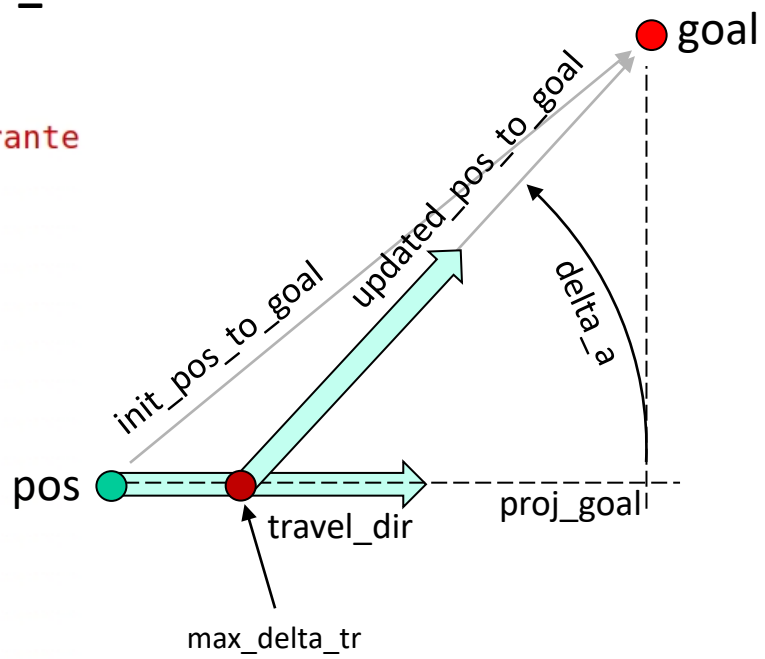


Exemple: spécialisation d'une méthode code source rob.cc (3)

```
typedef double Orient;

class Dir: public Rob // robot directionnel, doit tourner
{
public:
    Dir():Rob(),a(0){}
    void move_to(S2d goal); // avance puis tourne (avec limitations)
private:
    Orient a; // angle représentation l'orientation en rd
};
constexpr double max_delta_rt(0.8);
```

```
void Dir::move_to(S2d goal)
{
    // mise à jour de la position avec un déplacement selon l'orientation courante
    S2d    init_pos_to_goal = {goal[X] - pos[X], goal[Y] - pos[Y]};
    S2d    travel_dir    = {cos(a), sin(a)}; //vecteur unitaire selon Xrobot
    double proj_goal= s2d_prod_scal(init_pos_to_goal, travel_dir);

    if(abs(proj_goal) > max_delta_tr)
        proj_goal = ((proj_goal > 0) ? 1 : -1)*max_delta_tr;

    s2d_add_scaled_vector(pos, travel_dir, proj_goal);

    // mise à jour de l'orientation
    S2d    updated_pos_to_goal = {goal[X] - pos[X], goal[Y] - pos[Y]};
    Orient goal_a(atan2(updated_pos_to_goal[Y],updated_pos_to_goal[X]));
    Orient delta_a(goal_a - a);

    if(abs(delta_a) <= max_delta_rt) a = goal_a ;
    else  a += ((delta_a > 0) ?  1. : -1.)*max_delta_rt ;
}
```

Pour la classe **Dir** la méthode **move_to** spécialisée déplace d'abord le robot selon sa direction courante (Xr) mais au plus d'une distance **max_delta_tr** pour atteindre la projection du point **goal** sur la direction courante, et modifie son orientation pour s'aligner avec la mise à jour du vecteur **pos_to_goal** mais au plus de **max_delta_rt.**



Exemple: spécialisation d'une méthode code source rob.cc (4)

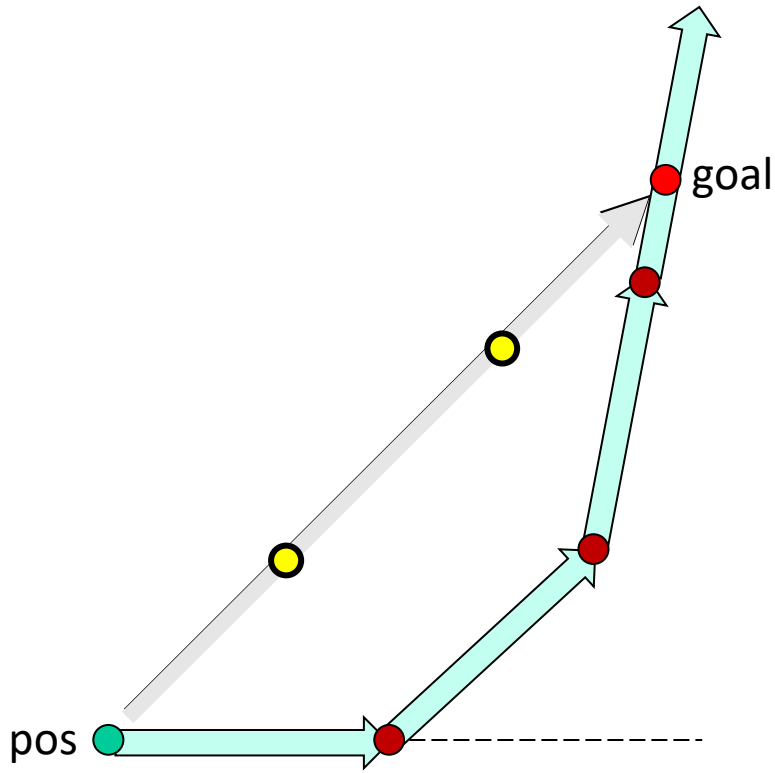
test de la classe

```
int main()
{   // les 3 robots sont initialisés à l'origine (0,0)
    Rob a;
    Omni b;
    Dir c;

    // téléportation
    a.affiche();
    a.move_to({10., 10.});
    a.affiche();
    cout << endl;

    // omnidirectionnel
    b.affiche();
    for(int i(0) ; i<3 ; ++i)
    {
        b.move_to({10., 10.});
        b.affiche();
    }
    cout << endl;

    // non-holonome
    c.affiche();
    for(int i(0) ; i<4 ; ++i)
    {
        c.move_to({10., 10.});
        c.affiche();
    }
    cout << endl;
    return 0;
}
```

```
0       0
10      10
```

```
0       0
3.53553 3.53553
7.07107 7.07107
10      10
```

```
0       0
5       0
8.48353 3.58678
9.6341  8.4526
10      10
```



Quizz2: surcharge & masquage

prog.cc

```
1  #include <iostream>
2  #include <string>
3  class A {
4  public:
5      void m1(int i)          {std::cout << "A::m1(int)"    << i << std::endl;}
6      void m1(std::string s) {std::cout << "A::m1(string)" << s << std::endl;}
7  };
8  class B: public A {
9  public:
10     void m1(std::string s) {std::cout << "B::m1(string)" << s << std::endl;}
11     void h(int i)           { m1(i); }
12 };
```

La compilation (option –c) du fichier source **`prog.cc`** produit...

A un fichier **`prog.o`** sans aucun warning

B une erreur à la ligne 10 car **`m1`** est déjà définie par la classe A

C une erreur à la ligne 11 à cause de l'appel de **`m1`**

D une erreur pour une autre raison

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
}
```

Quizz3:

<u>Question Hiérachie de classes (MOOC semaine 4)</u>:
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