LANGAGE C++

Le polymorphisme

Quelques rappels sur l'héritage

Dans une hiérarchie de classes, la sous-classe hérite de la super-classe :

- tous les attributs/méthodes publics ou protégés (sauf constructeurs et destructeur)
- le type :
  - ✓ on peut affecter un objet de type sous-classe à une variable de type super-classe :
  - ✓ passer en paramètre un objet d'une sous classe

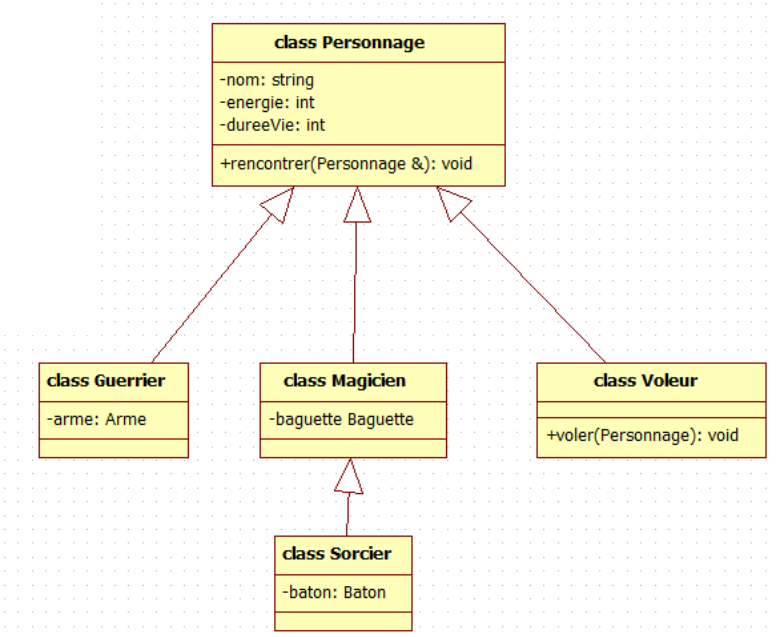
```
Personnage p;
Guerrier g;
// ...
p = g;
```

```
void faire_rencontrer(Personnage &un, Personnage  &autre)
{
    cout << un.Getnom() << " rencontre " << autre.Getnom() << " : " ;
    un.rencontrer(autre) ;
}

int main()
{
    cout << "Polymorphisme!" << endl;
    Guerrier guerrier("Labrute");
    Voleur voleur("Lerenard");
    faire_rencontrer(guerrier, voleur);
    return 0;
}
```

L'héritage est transitif

Un Sorcier est un Magicien qui est un Personnage



Polymorphisme (universel d'inclusion) :

- Grâce à l'héritage, le même code pourra être appliqué à un Magicien, un Guerrier, ... qui sont des **Personnage** .

- La façon dont un Personnage en rencontre un autre peut prendre plusieurs formes : le saluer (Magicien), le frapper (Guerrier), le voler (Voleur)...

- Grâce au polymorphisme, le même code appliqué à différents personnages pourra avoir un comportement différent, propre à chacun.

En POO, le **polymorphisme** (universel d'inclusion) est le fait que les instances d'une sous-classe, lesquelles sont substituables aux instances des classes de leur ascendance (en argument d'une méthode, lors d'affectations), **gardent leurs propriétés propres**.

Le choix des méthodes à invoquer se fait lors de l'exécution du programme en fonction de la nature réelle des instances concernées.

La mise en œuvre se fait au travers de :
- l'héritage (hiérarchies de classes) ;
- **la résolution dynamique des liens**.

Résolution des liens :

Quel est la méthode rencontrer appelée ?

- Celle de la classe Personnage
- Celle de la classe Guerrier

```
void faire_rencontrer(Personnage &un, Personnage  &autre)
{
    cout << un.Getnom() << " rencontre " << autre.Getnom() << " : " ;
    un.rencontrer(autre) ;
}

int main()
{
    cout << "Polymorphisme!" << endl;
    Guerrier guerrier("Labrute");
    Voleur voleur("Lerenard");
    faire_rencontrer(guerrier, voleur);
    return 0;
}
```

```
Polymorphisme!
Labrute rencontre Lerenard : Rencontrer de personnage
Process returned 0 (0x0)   execution time : 0.023 s
Press any key to continue.
```

```
class Personnage
{
    public:
        Personnage(string name):nom(name){};
         ~Personnage();
        string Getnom() { return nom; }
         void rencontrer(Personnage &p)
        {
            cout << "Rencontrer de personnage";
        }
    private:
        string nom;
};

   class Voleur : public Personnage
{
    public:
        Voleur(string name):Personnage(name){};
         ~Voleur();

         void rencontrer(Personnage &p)
        {
            cout << "Rencontrer de voleur";
        }
};

    class Guerrier : public Personnage
{
    public:
        Guerrier(string name):Personnage(name){};
        ~Guerrier();
        void rencontrer(Personnage &p)
        {
            cout << "Rencontrer de guerrier";
        }

};
```

Résolution des liens :

- **Résolution statique**

En C++, par défaut, c'est le type de la variable qui détermine la méthode à exécuter. Le choix de la méthode à exécuter se fait à la compilation.

- **Résolution dynamique des liens**

Le type effectif (type de l'objet en mémoire) qui détermine la méthode à exécuter. Pour mettre en œuvre le polymorphisme, il faut permettre la résolution dynamique des liens. Le choix de la méthode à exécuter se fait à l'exécution, en fonction de la nature réelle des instances

2 ingrédients pour cela : **références/pointeurs et méthodes virtuelles,**

Déclaration des méthodes virtuelles :

En C++, on indique au compilateur qu'une méthode peut faire l'objet d'une résolution dynamique des liens en la déclarant comme virtuelle (mot clé **virtual**)

- Cette déclaration doit se faire dans la classe la **plus générale** qui admet cette méthode (c'est-à-dire lors du prototypage d'origine)
- Les **redéfinitions** éventuelles dans les sous-classes seront aussi considérées comme virtuelles par **transitivité**.

Syntaxe :

```
virtual Type nom_fonction(liste de paramètres) [const];
```

Exemple :

```
class Personnage {
    // ...
    virtual void rencontrer(Personnage& p) {
        cout << "Saluer";
    }
};
```

Exemple concret :

```
Labrute rencontre Lerenard : Renconrer de Guerrier
Process returned 0 (0x0)   execution time : 0.020 s
Press any key to continue.
```

```
class Personnage {
    public:
        Personnage(string n) : nom(n)
        {}
        virtual void rencontrer(Personnage& p) {
            cout << "Rencontrer de Personnage";
        }
        string getNom()const { return nom; }
    private: string nom;
};
class Voleur : public Personnage {
    public:
        Voleur(string nom) : Personnage(nom)
        {}
        void rencontrer(Personnage& p)  {
            cout << "Rencontrer de Voleur";
        }
};
```

```
class Guerrier : public Personnage {
    public:
        Guerrier(string nom) : Personnage(nom)
        {}
        void rencontrer(Personnage& p)  {
            cout << "Renconrer de Guerrier";
        }
};
void faire_rencontrer(Personnage & un, Personnage & autre) {
    cout << un.getNom() << " rencontre " << autre.getNom() << " : ";
    un.rencontrer(autre);
}
int main() {
    Guerrier guerrier("Labrute"); Voleur voleur("Lerenard");
    faire_rencontrer(guerrier, voleur);
    return 0;
}
```

Exemple concret :

attention !!

```
void faire_rencontrer(Personnage un, Personnage autre) {
    cout << un.getNom() << " rencontre " << autre.getNom() << " : ";
    un.rencontrer(autre);
}
```

Le paramètre un étant passé par **valeur**, la valeur effective de l'objet un est de type **Personnage**.

```
Labrute rencontre Lerenard : Rencontrer de personnage
Process returned 0 (0x0)   execution time : 0.023 s
Press any key to continue.
```

Autre exemple :

```
class Mammifere {
    public:
        Mammifere() { cout << "Un nouveau mammifere est ne !" << endl; }
        virtual ~Mammifere() { cout << "Un mammifere est en train de mourir :(" << endl; }
        void manger() const { cout << "Miam... croumf !" << endl; }
        virtual void avancer() const { cout << "Un grand pas pour l'humanite." << endl; }
};
class Dauphin : public Mammifere {
    public:
        Dauphin () { cout << "Coui, Couic !" << endl; }
        ~Dauphin() { cout << "Flipper, c'est fini..." << endl; }
        void manger() const { cout << "Sglups, un poisson." << endl; }
        void avancer() const { cout << "Je nage." << endl; }
};
int main() {
    Mammifere* lui(new Dauphin());
    lui->avancer();
    lui->manger();
    delete lui;
    return 0;
}
```

```
Un nouveau mammifère est né !          Mammifere::Mammifere()
Coui, Couic !                          Dauphin::Dauphin()
Je nage.                               Dauphin::avancer()
Miam... croumf !                       Mammifere::manger()
Flipper, c'est fini...                 Dauphin::~Dauphin()
Un mammifère est en train de mourir :(  Mammifere::~Mammifere()
```

Si le destructeur de Mammifère n'avait pas été virtuel ?

```
Un nouveau mammifère est né !
Coui, Couic !
Je nage.
Miam... croumf !
Un mammifère est en train de mourir :(
```

Pas d'appel au destructeur de Dauphin.
(si l'objet Dauphin avait alloué des ressources, elles n'auraient pas été désallouées)

Autre exemple :

Lorsqu'une méthode virtuelle est invoquée à partir d'une référence ou d'un pointeur vers une instance, c'est la méthode du type réel de l'instance qui sera exécutée.

Attention !

- Il est conseillé de toujours définir les destructeurs comme virtuels
- Un constructeur ne peut pas être virtuel
- L'aspect virtuel des méthodes est ignoré dans les constructeurs, c'est la méthode de la classe courante qui est appelée.

Masquage, substitution et surcharge :

Trois concepts différents :

- la surcharge (**overloading**) de fonctions et de méthodes
- le masquage (**shadowing**) (en particulier de méthodes)
- (sans la nommer jusqu'ici) la substitution (**overriding**), dans les sous-classes, de nouvelles versions de méthodes virtuelles.

Par ailleurs, C++ 2011, introduit deux nouveaux mots clés, optionnels, pour justement aider le programmeur à préciser ses intentions : **override** et **final**

surcharge : même nom, mais paramètres différents, (en C++, il ne peut y avoir surcharge que dans la même portée).

masquage : entités de mêmes noms mais de portées différentes, masqués par les règles de résolution de portée.
Attention aux subtilités : une seule méthode de même nom suffit à les masquer toutes, indépendamment des paramètres !

substitution des méthodes virtuelles :

- Résolution dynamique : c'est la méthode de l'instance qui est appelée (si pointeur ou référence)
- Si l'on redéfinit qu'une seule méthode (virtuelle) surchargée, alors les autres sont masquées

Masquage et surcharge - exemple :

```
class A {
    public:
        virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
        // surcharge :
        virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};
class B : public A {
    public:
        // substitution de l'une des deux, l'autre devient hors de portée (masquage)
        void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};
class C : public A {
    public:
        // introduction d'une 3e => masquage des 2 autres
        void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
};
```

```
A::m1(int) : 2
B::m1(string)
C::m1(double) : 2
A::m1(string) : 2
A::m1(int) : 2

Process returned 0 (0x0)
Press any key to continue.
```

```
int main()
{
    B b;
    //b.m1(2);      // NON : no matching function for call to 'B::m1(int)'
    b.A::m1(2);     // ... mais elle est bien là
    b.m1("2");
    C c;
    c.m1(2);        // Attention ici : c'est celle avec double !!
    //c.m1("2");    // NON : no matching function
    c.A::m1("2");   // OK
    c.A::m1(2);     // OK, et là c'est celle avec int
    return 0;
}
```

Substitution – exemple

```
class A {
    public:
        virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
        // surcharge :
        virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};
class B : public A {
    public:
        // substitution de l'une des deux, l'autre devient hors de portée (masquage)
        void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};
class C : public A {
    public:
        // introduction d'une 3e => masquage des 2 autres
        void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
};
```

```
int main()
{
    B b;
    C c;
    A* pa(nullptr);
    pa = &b;
    pa->m1("2");
    pa->m1(2);          // OK (nous sommes dans A::)
    pa = &c;
    pa->m1(2.1);        // Attention ici : c'est celle avec int !!
                        // Nous sommes dans A::
    // pa->C::m1(2.1);  // Impossible ! A n'hérite pas de C !!
    return 0;
}
```

```
B::m1(string)
A::m1(int) : 2
A::m1(int) : 2

Process returned 0 (0x0)
Press any key to continue.
```