

Cours d'Informatique C++

Partie :Polymorphisme

Professeur: Ansam ENNACIRI

Polymorphisme



Quelques rappels sur l'héritage

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

- tous les attributs/méthodes (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

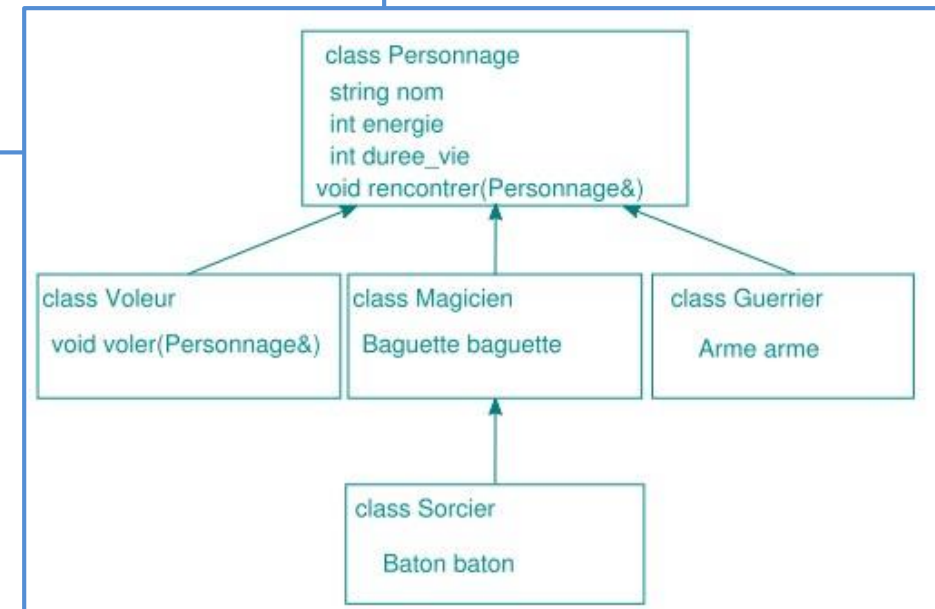
```
Personnage p;
Guerrier g;
// ...
p = g;
```

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

int main() {
    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.

Aller vers du code générique : Code qui s'écrit de façon unifié pour différent type de données

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

```
class Guerrier : public Personnage {
public:
    Guerrier(string nom) : Personnage(nom)
    {}
    void rencontrer(Personnage& p) {
        cout << "Rencontrer 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;
}
```

```
class Personnage {
public:
    Personnage(string n) : nom(n)
    {}
    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";
    }
};
```

Résolution statique

En C++, **par défaut**, c'est le type de la variable qui détermine la méthode à exécuter :

=> **résolution statique des liens**

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 complet

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

```
class Guerrier : public Personnage {
public:
    Guerrier(string nom) : Personnage(nom)
    {}
    void rencontrer(Personnage& p) {
        cout << "Rencontrer 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;
}
```

```
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";
    }
};
```


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**.

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)

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

Méthodes virtuelles et constructeurs

L'aspect polymorphique est ignoré dans les constructeurs ; c'est la méthode de la classe courante qui est appelée.

```
class A {  
    public:  
        A() { f(); }  
        virtual void f() const { cout << "A::f()" << endl; }  
};  
class B : public A {  
    public:  
        void f() const { cout << "B::f()" << endl; }  
};  
int main()  
{  
    A a;  
    B b;  
    A* pa(&b);  
    pa->f();  
    return 0;  
}
```

```
A::f(<)  
A::f(<)  
B::f(<  
  
Process returned 0 (0x0)  
Press any key to continue.
```

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;
}
```

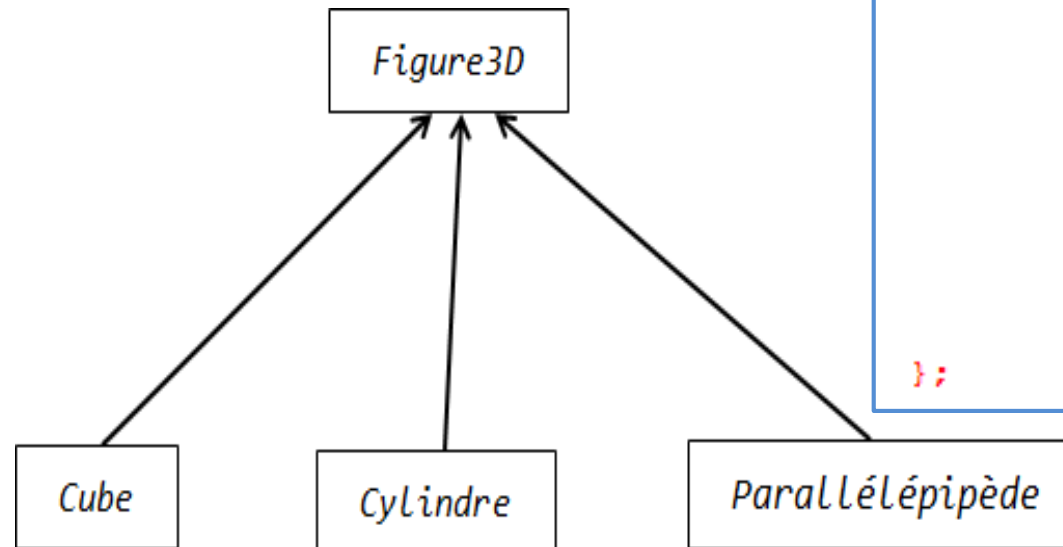
```
B::m1(string)
A::m1(int) : 2
A::m1(int) : 2

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

Méthodes virtuelles pures – contexte

Au sommet d'une hiérarchie de classe, il n'est pas toujours possible de :

- donner une définition générale de certaines méthodes, compatibles avec toutes les sous-classes,
- ...même si l'on sait que toutes ces sous-classes vont effectivement implémenter ces méthodes



```

class Figure3D {
    private:
        double hauteur;
    public:
        // difficile à définir à ce niveau !..
        virtual double surface() const {
            ????????
        }
        // La méthode volume en aurait besoin !
        double volume() const {
            return hauteur * surface();
        }
};
  
```

La méthode surface de la classe Figure3D n'a pas de sens



Déclarer la méthode surface comme **virtuelle pure**

Besoin de méthodes virtuelles pures

Classe pour gérer le jeu (se contente ici d'afficher les personnages)

Nous ne savons pas comment afficher un personnage générique (classe Personnage) tandis que nous savons afficher un personnage « spécialisé »

```
class Personnage {
private:
    string nom;
};
class Jeu {
public:
    // ...
    void afficher() const {
        for (auto un_perso : persos) {
            un_perso->afficher();
            // Tous les personnages doivent pouvoir s'afficher !...
            // ...mais comment ???
        }
    }
    // ...
private:
    vector<Personnage*> persos;
};
```

Message

```
=== Build: Debug in heritagel (compiler: GNU GCC Compiler) ===
warning: "/*" within comment [-Wcomment]
In member function 'void Jeu::afficher() const':
error: 'class Personnage' has no member named 'afficher'
=== Build failed: 1 error(s), 1 warning(s) (0 minute(s), 0 second(s))
```

De plus, on aimerait :

- **imposer** aux sous-classes (Guerrier , ...) d'avoir **leur méthode** afficher **spécifique**
- que cette méthode spécifique à la sous-classe soit polymorphique => donc méthode **virtuelle**

=> Déclarer la méthode comme virtuelle pure

Méthodes virtuelles pures - définition et syntaxe

Une méthode virtuelle pure, ou abstraite :

- sert à imposer aux sous-classes (non abstraites) qu'elles doivent redéfinir la méthode virtuelle héritée
- est signalée par un `= 0` en fin de prototype,
- est, en général, incomplètement spécifiée : il n'y a très souvent pas de définition dans la classe où elle est introduite (pas de corps).

Syntaxe

```
virtual Type nom_methode(liste de paramètres) = 0;
```

Exemple

```
class Personnage {  
    // ...  
public:  
    virtual void afficher() const = 0;  
    // ...  
};
```

Autre

```
class Figure3D {  
private:  
    double hauteur;  
public:  
    virtual double surface() const = 0;  
    // On peut utiliser une méthode virtuelle pure :  
    double volume() const {  
        return hauteur * surface();  
    }  
};
```

Exemple complet

```
Cube s = 21.16 v = 97.336
Cylindre s = 15.2053 v = 97.3142
Parallelepipede s = 36.12 v = 245.616

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

```
class Cylindre : public Figure3D {
    // ...
};
class Parallelepipede : public Figure3D {
    // ...
};
int main()
{
    Cube cube(4.6);
    cout << "Cube s = " << cube.surface() << " v = " << cube.volume() << endl;
    Cylindre cylindre(2.2, 6.4);
    cout << "Cylindre s = " << cylindre.surface() << " v = " << cylindre.volume() << endl;
    Parallelepipede Parallel(4.2, 8.6, 6.8);
    cout << "Parallelepipede s = " << Parallel.surface() << " v = " << Parallel.volume() << endl;
    return 0;
}
```

```
class Figure3D {
private:
    double hauteur;
public:
    Figure3D(double h) : hauteur(h)
    {}
    virtual double surface() const = 0;
    // On peut utiliser une méthode virtuelle pure :
    double volume() const {
        return hauteur * surface();
    }
};
class Cube : public Figure3D {
private:
    double cote;
public:
    Cube(double arrete) : Figure3D(arrete), cote(arrete)
    {}
    double surface() const {
        return cote*cote;
    }
};
```

Classes abstraites

Une **classe abstraite** est une classe contenant **au moins une méthode virtuelle pure**.

- Elle ne peut être instanciée
- Ses sous-classes restent abstraites tant qu'elles ne fournissent pas les définitions de toutes les méthodes virtuelles pures dont elles héritent. (En toute rigueur : tant qu'elles ne suppriment pas l'aspect virtuel pur (le « = 0 »).)

```
class Personnage {
public:
    Personnage(string n) : nom(n)
    {}
    virtual void afficher() const = 0;
    string getNom() const { return nom; }
private: string nom;
};

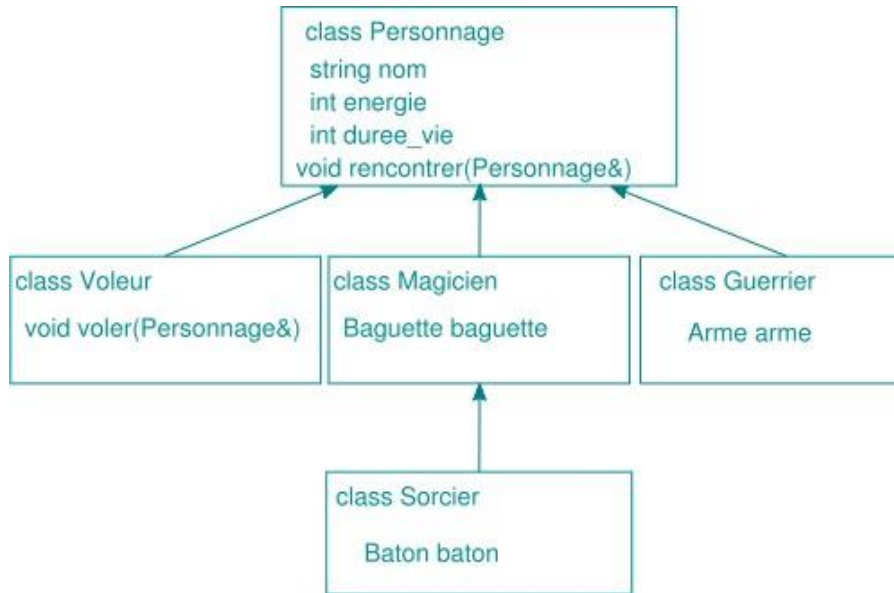
class Guerrier : public Personnage {
public:
    Guerrier(string nom) : Personnage(nom)
    {}
};
```

```
int main()
{
    Guerrier guerrier("Labrute");
    return 0;
}
```

```
Message
=== Build: Debug in heritagel (compiler: GNU GCC Compiler) ===
In function 'int main()':
error: cannot declare variable 'guerrier' to be of abstract type 'Guerrier'
note: because the following virtual functions are pure within 'Guerrier':
note: virtual void Personnage::afficher()
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Collection hétérogène

- L'héritage et les méthodes virtuelles permettent de mettre en œuvre des **traitements génériques** sur les instances d'une hiérarchie de classes (**polymorphisme d'inclusion**).
- Les fonctions/méthodes génériques doivent utiliser des arguments passés par référence ou pointeur pour que le traitement se fasse en fonction de la nature réelle de l'instance



Pour pouvoir gérer le déroulement du jeu nous devons avoir une liste de personnages qui est une liste d'objets de type différent (Magicien, Voleur,...) mais appartenant à la même hiérarchie de classes.

=> Collection hétérogène

(au sens où le comportement spécifique de chaque instance de la collection peut être différent)

- Permet de gérer tous les personnages de façon globale (générique),
- tous les personnages ont leurs spécificités propres (comportements).

On pourrait par exemple souhaiter plutôt écrire quelque chose comme :

```
vector<Personnage> personnages;
```

ne permet **pas** le comportement polymorphique :

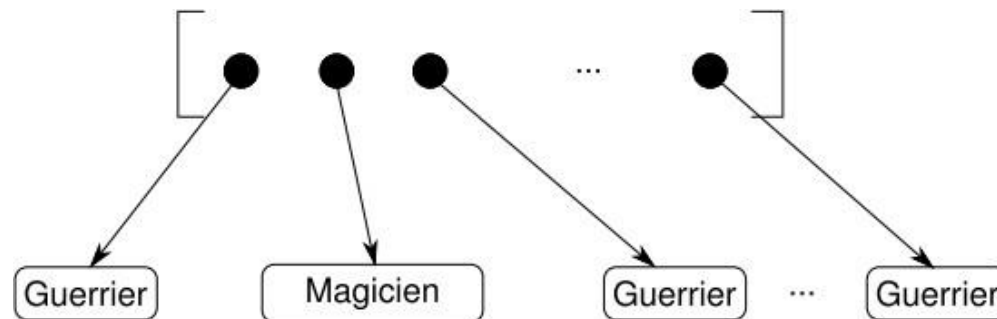
Le vecteur personnages est constitué d'instances de type Personnage et non pas de références/pointeurs à ces instances.

Collection hétérogène et polymorphisme

Si l'on veut une collection avec comportement polymorphique des éléments,
il faut une collection de **pointeurs**

```
vector<Personnage*> personnages;
```

Seuls les pointeurs, c'est-à-dire les adresses des instances, sont stocké(e)s dans la collection, et non plus les instances elles-mêmes :



Exemple complet

```
class Personnage {
public:
    Personnage(string n) : nom(n)
    {}
    virtual void afficher() const = 0;
    string getNom() const { return nom; }
private: string nom;
};

class Guerrier : public Personnage {
public:
    Guerrier(string nom) : Personnage(nom)
    {}
    void afficher() const {
        cout << "Guerrier " << getNom() << endl;
    }
};

class Voleur : public Personnage {
public:
    Voleur(string nom) : Personnage(nom)
    {}
    void afficher() const {
        cout << "Voleur " << getNom() << endl;
    }
};
```

```
Guerrier Labrute
Voleur Laruse
Guerrier Lecolosse
Magicien Leprestigitateur
Sorcier Lebalaie
```

```
class Magicien : public Personnage {
public:
    Magicien(string nom) : Personnage(nom)
    {}
    void afficher() const {
        cout << "Magicien " << getNom() << endl;
    }
};

class Sorcier : public Magicien {
public:
    Sorcier(string nom) : Magicien(nom)
    {}
    void afficher() const {
        cout << "Sorcier " << getNom() << endl;
    }
};

int main()
{
    vector<Personnage*> personnages;
    personnages.push_back(new Guerrier("Labrute"));
    personnages.push_back(new Voleur("Laruse"));
    personnages.push_back(new Guerrier("Lecolosse"));
    personnages.push_back(new Magicien("Leprestigitateur"));
    personnages.push_back(new Sorcier("Lebalaie"));

    for (auto const& quidam : personnages) {
        quidam->afficher();
    }

    return 0;
}
```

Exemple avec classe jeu

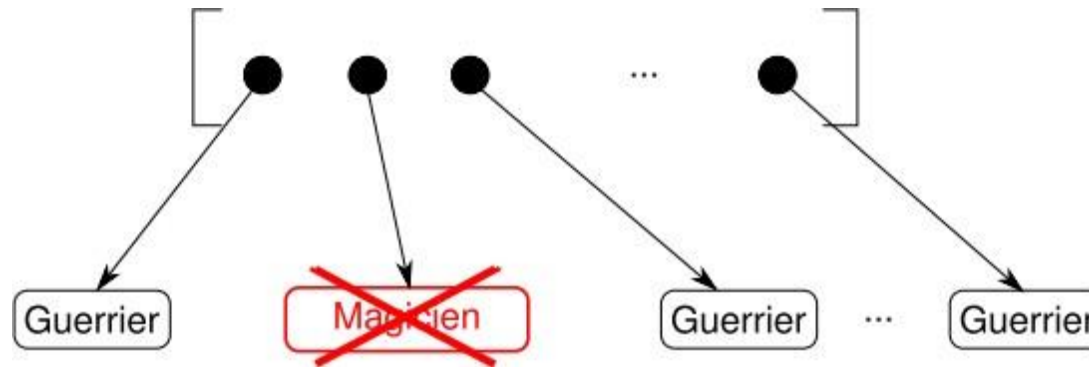
```
class Jeu {  
public:  
    void afficherTousPersos() const {  
        for (auto un_perso : persos) {  
            un_perso->afficher();  
        }  
    }  
    void ajouterPersonnage(Personnage *perso) {  
        if (perso != nullptr) {  
            persos.push_back(perso);  
        }  
    }  
private:  
    vector<Personnage*> persos;  
};
```

```
Guerrier Labrute  
Voleur Laruse  
Guerrier Lecolosse  
Magicien Leprestigiteur  
Voleur Lechourave  
Sorcier Lebalaie  
  
Process returned 0 (0x0)  
Press any key to continue.
```

```
int main()  
{  
    Jeu jeu;  
    jeu.ajouterPersonnage(new Guerrier("Labrute"));  
    jeu.ajouterPersonnage(new Voleur("Laruse"));  
    jeu.ajouterPersonnage(new Guerrier("Lecolosse"));  
    jeu.ajouterPersonnage(new Magicien("Leprestigiteur"));  
    jeu.ajouterPersonnage(new Voleur("Lechourave"));  
    jeu.ajouterPersonnage(new Sorcier("Lebalaie"));  
    jeu.afficherTousPersos();  
  
    return 0;  
}
```


Pointeurs et intégrité des données

Pour que tout fonctionne bien, il est nécessaire que les éléments pointés existent **aussi longtemps** que leurs pointeurs.



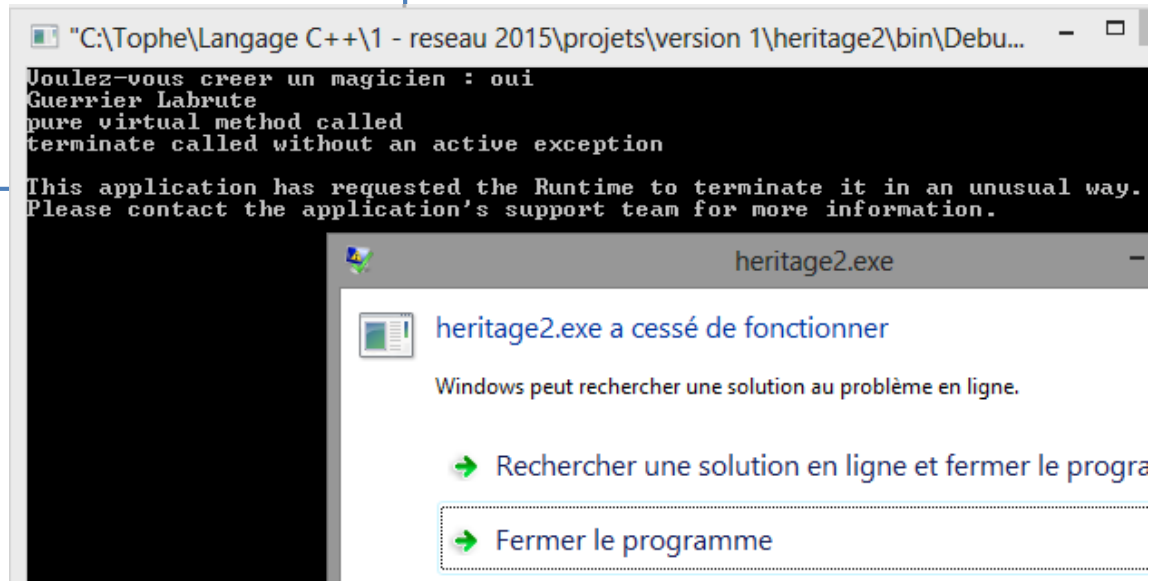
Attention ! La coexistence des pointeurs et des éléments pointés n'est cependant pas du tout garantie !
Elle est de la responsabilité du programmeur.

Dans l'exemple de la page suivante

- Le Magicien manipulateur est créé statiquement dans un bloc (variable locale),
- Cette variable sera détruite à la sortie du bloc (désallocation mémoire)
- L'adresse de la variable manipulateur est toujours stockée dans le vecteur personnage
- Dans la structure itérative on appelle la méthode afficher d'un objet qui a été désallouer

Exemple

```
int main()
{
    string reponse = "non";
    vector<Personnage*> personnages;
    personnages.push_back(new Guerrier("Labrute"));
    cout << "Voulez-vous creer un magicien : ";
    cin >> reponse;
    if(reponse == "oui"){
        Magicien manipulateur("Lemanipulateur");
        personnages.push_back(&manipulateur);
    }
    for (auto quidam : personnages) {
        quidam->afficher();
    }
    return 0;
}
```



Gare aux pointeurs !

L'utilisation des « pointeurs intelligents » `unique_ptr` présente deux avantages :

1. pas besoin de se préoccuper de la désallocation
2. l'aspect « unique » évite les références multiples et leur gestion/cohérence

=> On a beaucoup moins de précautions à prendre et de garde-fous à programmer !

Qui dit « pointeurs C », dit aussi si nécessaire :

=> copie profonde, destructeur pour libérer la mémoire allouée et opérateur d'affectation.

Désallocation mémoire

Le développeur qui a alloué la mémoire (**new**) à la responsabilité de la libérer (**delete**)

```
void detruiteTousPersos(){  
    for (auto quidam : personnages) {  
        delete quidam;  
    }  
    personnages.clear();  
}
```

```
void detruirePersonnage(size_t lequel)  
{  
    delete personnages[lequel];  
    // puis, en fonction des situations :  
    // SOIT  
    // préserve les index et la taille de la collection  
    personnages[lequel] = nullptr;  
    // SOIT  
    // suppression efficace mais ne préserve pas l'ordre  
    swap(personnages[lequel], personnages.back());  
    personnages.pop_back();  
    // SOIT  
    // suppression plus coûteuse qui préserve l'ordre  
    personnages.erase(personnages.begin() + lequel);  
}
```

Exemple

```
class Jeu {
public:
    void afficherTousPersos() const {
        for (auto perso : personnages) {
            if (perso != nullptr) {
                perso->afficher();
            }
        }
    }
    void ajouterPersonnage(Personnage *perso) {
        if (perso != nullptr) {
            personnages.push_back(perso);
        }
    }
    void detruiteTousPersos(){
        for (auto quidam : personnages) {
            delete quidam;
        }
        personnages.clear();
    }
    void detruirePersonnage(size_t lequel)
    {
        delete personnages[lequel];
        personnages[lequel] = nullptr;
    }
private:
    vector<Personnage*> personnages;
};
```

```
int main()
{
    Jeu jeu;
    jeu.ajouterPersonnage(new Guerrier("Labrute"));
    jeu.ajouterPersonnage(new Voleur("Laruse"));
    jeu.ajouterPersonnage(new Guerrier("Lecolosse"));
    jeu.ajouterPersonnage(new Magicien("Leprestigitateur"));
    jeu.ajouterPersonnage(new Voleur("Lechourave"));
    jeu.ajouterPersonnage(new Sorcier("Lebalaie"));
    cout << "INITIALISATION PERSONNAGE" << endl;
    jeu.afficherTousPersos();
    jeu.detruirePersonnage(2);
    cout << endl << "DESTRUCTION GUERRIER Lecolosse" << endl;
    jeu.afficherTousPersos();
    jeu.detruiteTousPersos();
    cout << endl << "DESTRUCTION TOUS PERSONNAGES" << endl;
    jeu.afficherTousPersos();

    return 0;
}
```

```
INITIALISATION PERSONNAGE
Guerrier Labrute
Voleur Laruse
Guerrier Lecolosse
Magicien Leprestigitateur
Voleur Lechourave
Sorcier Lebalaie

DESTRUCTION GUERRIER Lecolosse
Guerrier Labrute
Voleur Laruse
Magicien Leprestigitateur
Voleur Lechourave
Sorcier Lebalaie

DESTRUCTION TOUS PERSONNAGES

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