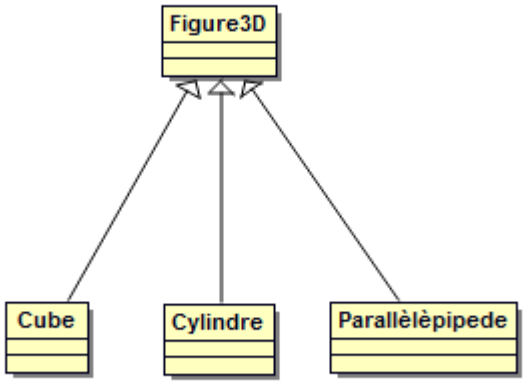
LANGAGE C++

Les classes abstraites & les collections hétérogènes

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 vomume en aurait besoin !
        double volume() const {
            return hauteur * surface();
        }
};
```

La méthode surface() de la classe Figure3D n'as pas de sens

=> Il faut la déclarer 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 »

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.
```

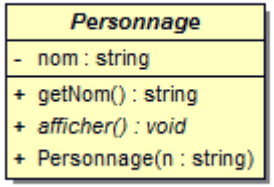
```cpp
class Figure3D {
    private:
        double hauteur;
    public:
        Figure3D(double h) : hauteur(h)
        {}
        virtual double surface() const = 0;
        // On peut utiliser une méthode virtuelle pure :
        double volume() const {
            return hauteur * surface();
   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

```cpp
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;
}
```

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 »).)



Le nom d'une classe abstraite est écrit en italique.

La méthode virtuelle pure peut également être écrite en italique ou écrite + afficher() =0 : void

```
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 heritage1 (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)) ===
```

Classes abstraites

On peut résumer les fonctions virtuelles de la manière suivante :

- une méthode virtuelle *peut* être redéfinie dans une classe fille ;
- une méthode virtuelle pure *doit* être redéfinie dans une classe fille.

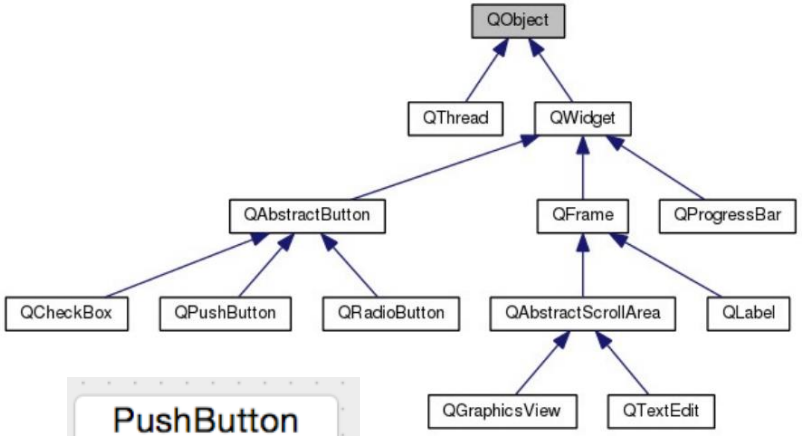
Dans la bibliothèque **Qt**, que nous allons très bientôt aborder, il y a beaucoup de classes abstraites.

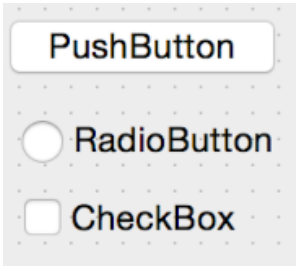
Il existe par exemple une classe par sorte de bouton, c'est-à-dire une classe pour les boutons normaux, une pour les cases à cocher, etc.

Toutes ces classes héritent d'une classe nommée *QAbstractButton*, qui regroupe des propriétés communes à tous les boutons (taille, texte, etc.).

Mais comme on ne veut pas autoriser les utilisateurs à mettre des QAbstractButton sur leurs fenêtres, les créateurs de la bibliothèque ont rendu cette classe abstraite.

Arbre d'héritage (très partiel)

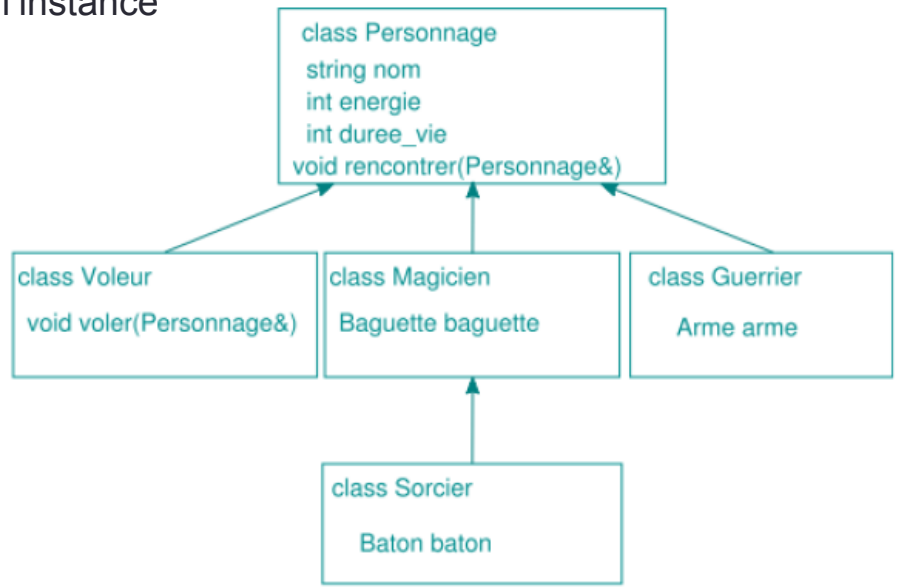




Collection hétérogène

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 :

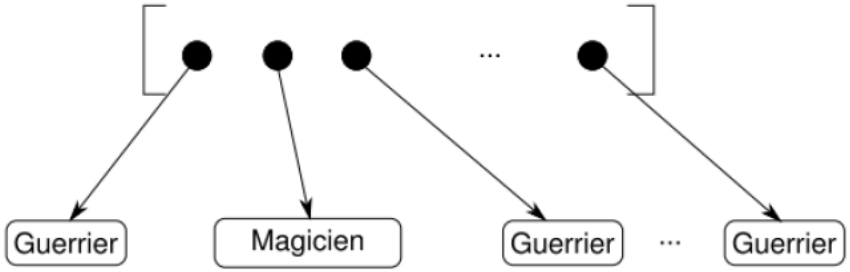
Le vecteur personnages est constitué d'instances de type Personnage et non pas de références/pointeurs à ces instances.

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

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

```
class Personnage
{
    public:
        Personnage(string name):nom(name){};
         ~Personnage();
        string Getnom() { return nom; }
        virtual void rencontrer(Personnage &p)
        {
            cout << "Rencontrer de personnage";
        }
        virtual void afficher()=0;
    private:
        string nom;
};
```

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

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

             void rencontrer(Personnage &p)
            {
                cout << "Rencontrer de voleur";
            }
             void afficher()
            {
                cout << "Voleur " << Getnom() << endl;
            }
    };
```

```
class Magicien : public Personnage
{
    public:
         Magicien(string name):Personnage(name){};
        virtual ~Magicien();
        void rencontrer(Personnage &p)
        {
            cout << "Rencontrer de Magicien";
        }
        void afficher()
        {
            cout << "Magicien " << Getnom() << endl
        }

};
```

```
class Sorcier : public Magicien
{
    public:
        Sorcier(string name):Magicien(name){};
        virtual ~Sorcier();
        void rencontrer(Personnage &p)
        {
            cout << "Rencontrer de Sorcier";
        }
        void afficher()
        {
            cout << "Sorcier " << Getnom() << endl;
        }
};
```

```
Guerrier Labrute
Voleur Leruse
Guerrier Lecolosse
Magicien Leprestidigitateur
Sorcier Lebalai

Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.
```

```
int main()
{
    vector<Personnage * > personnages;

    personnages.push_back(new  Guerrier("Labrute"));
    personnages.push_back(new Voleur("Leruse"));
    personnages.push_back(new Guerrier("Lecolosse"));
    personnages.push_back(new Magicien("Leprestidigitateur"));
    personnages.push_back(new Sorcier("Lebalai"));

    for(int i=0; i< personnages.size(); i++)
    {
        personnages[i]->afficher() ;
    }
    return 0;
}
```

Exemple : unique_ptr C++11

unique_ptr : un seul pointeur peut référencer un espace mémoire

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


#include"voleur.h"
#include "guerrier.h"
#include "sorcier.h"
#include "magicien.h"


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



    int main()
    {
        vector<unique_ptr<Personnage>> personnages;

        personnages.push_back(unique_ptr<Personnage>(new  Guerrier("Labrute")));
        personnages.push_back(unique_ptr<Personnage>(new Voleur("Leruse")));
        personnages.push_back(unique_ptr<Personnage>(new Guerrier("Lecolosse")));
        personnages.push_back(unique_ptr<Personnage>(new Magicien("Leprestidigitateur")));
        personnages.push_back(unique_ptr<Personnage>(new Sorcier("Lebalai")));

        for(int i=0; i< personnages.size(); i++)
        {
            personnages[i]->afficher() ;
        }
        return 0;
    }
```

```
Guerrier Labrute
Voleur Leruse
Guerrier Lecolosse
Magicien Leprestidigitateur
Sorcier Lebalai

Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.
```

Exemple avec classe jeu

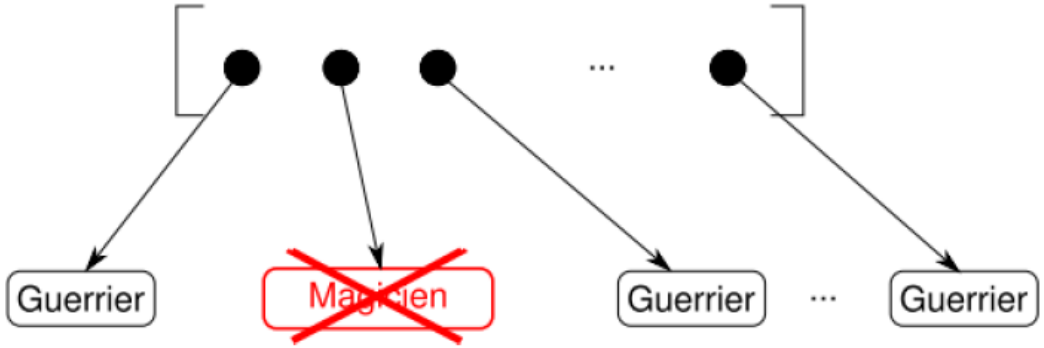
```cpp
#include <iostream>
#include <vector>
#include <memory>
using namespace std;

#include"voleur.h"
#include "guerrier.h"
#include "sorcier.h"
#include "magicien.h"


class Jeu
{
    public:
        Jeu();
        virtual ~Jeu();
        void afficherPersonnages() const
        {
             for(int i=0; i< personnages.size(); i++)
            {
                personnages[i]->afficher() ;
            }
        }
        void ajouterUnPersonnage(Personnage * perso)
        {
            if(perso != nullptr)
            {
                personnages.push_back(perso);
            }

        }

    private:
    vector<Personnage *> personnages;
};
```

```cpp
int main()
{
    Jeu jeu;
    jeu.ajouterUnPersonnage(new Guerrier("Labrute"));
    jeu.ajouterUnPersonnage(new Voleur("Leruse"));
    jeu.ajouterUnPersonnage(new Guerrier("Lecolosse"));
    jeu.ajouterUnPersonnage(new Magicien("Leprestidigitateur"));
    jeu.ajouterUnPersonnage(new Sorcier("Lebalai"));
    jeu.afficherPersonnages();

}
```

```
Guerrier Labrute
Voleur Leruse
Guerrier Lecolosse
Magicien Leprestidigitateur
Sorcier Lebalai

Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.
```

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.

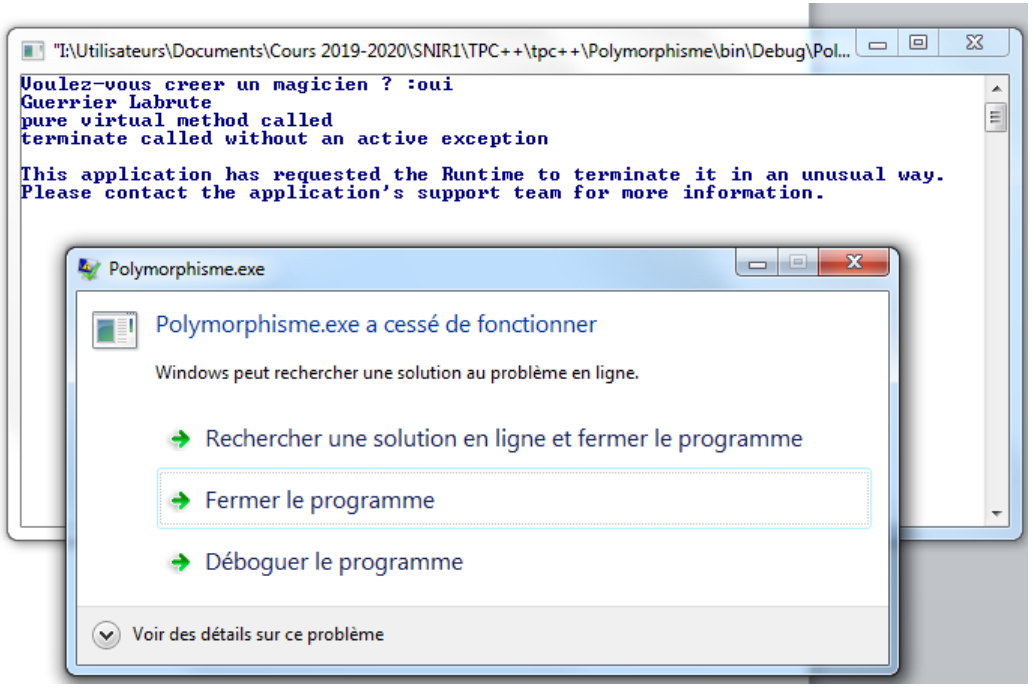
Dans l'exemple de la page suivante

- 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 désalloué

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(int i=0; i< personnages.size(); i++)
    {
            personnages[i]->afficher() ;
    }

    return 0;
}
```

- 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 désalloué



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 detruireTousLesPersonnages()
{
     for(int i=0; i< personnages.size(); i++)
    {
        delete personnages[i] ;
    }
    personnages.clear();
}
```

```
void detruirePersonnage(int 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 couteuse qui préserve l'ordre
    personnages.erase(personnages.begin()+lequel);
}
```

Exemple :

```
class Jeu
{
    public:
        Jeu();
        virtual ~Jeu();
        void afficherPersonnages() const
        {
             for(int i=0; i< personnages.size(); i++)
            {
                if(personnages[i] !=nullptr)
                {
                     personnages[i]->afficher() ;
                }
            }
        }
        void ajouterUnPersonnage(Personnage * perso)
        {
            if(perso != nullptr)
            {
                personnages.push_back(perso);
            }

        }
        void detruireTousLesPersonnages()
        {
             for(int i=0; i< personnages.size(); i++)
            {
                delete personnages[i] ;
            }
            personnages.clear();
        }
        void detruirePersonnage(int lequel)
        {
            delete personnages[lequel];
            personnages[lequel] = nullptr;
        }

    private:
    vector<Personnage *> personnages;
};
```

```
 int main()
{
    Jeu jeu;
    jeu.ajouterUnPersonnage(new Guerrier("Labrute"));
    jeu.ajouterUnPersonnage(new Voleur("Leruse"));
    jeu.ajouterUnPersonnage(new Guerrier("Lecolosse"));
    jeu.ajouterUnPersonnage(new Magicien("Leprestidigitateur"));
    jeu.ajouterUnPersonnage(new Sorcier("Lebalai"));
    cout << "INITIALISATION DES PERSONNAGES" << endl;
    jeu.afficherPersonnages();
    jeu.detruirePersonnage(2);
    cout << endl << "DESTRUCTION GUERRIER Lecolosse";
    jeu.afficherPersonnages();
    jeu.detruireTousLesPersonnages();
    cout << endl << "DESTRUCTION DE TOUS LES PERSONNAGES";
    jeu.afficherPersonnages();
    return 0;
}
```

```
INITIALISATION DES PERSONNAGES
Guerrier Labrute
Voleur Leruse
Guerrier Lecolosse
Magicien Leprestidigitateur
Sorcier Lebalai

DESTRUCTION GUERRIER LecolosseGuerrier Labrute
Voleur Leruse
Magicien Leprestidigitateur
Sorcier Lebalai

DESTRUCTION DE TOUS LES PERSONNAGES
Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

Pointeurs et intégrité des données

Problème potentiel avec des pointeurs « C » : **intégrité** des données

3 facettes :

1. durée de vie des données
2. désallocation
3. partage des données entre collections