Polymorphisme

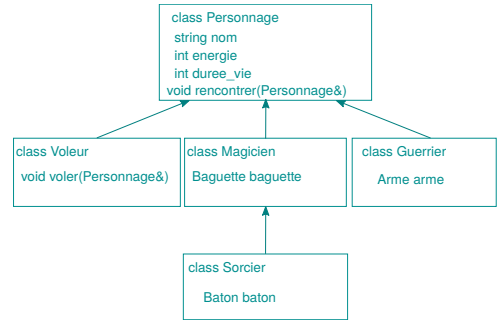
Concepts fondamentaux de l'orienté-objet :

- Encapsulation
- Abstraction
- Héritage
- Polymorphisme

Polymorphisme ?

Un exemple :

- 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.



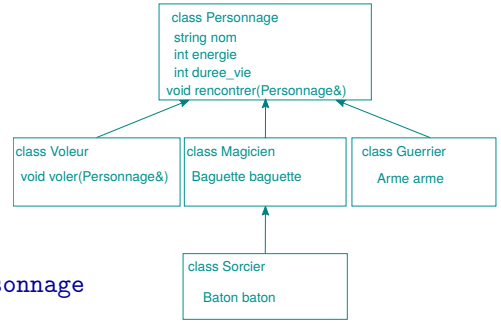

```
Personnage p1, p2;
// ...
p1.rencontrer(p2);
```

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 :

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



L'héritage est transitif :
un `Sorcier` est un `Magicien` qui est un `Personnage`

« est-un » : héritage du type

```
class Personnage {
public:
  // ...
  void rencontrer(Personnage& p) const
  { // ...
};


class Guerrier : public Personnage {
public:
  // ...


};
```

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

int main() {
  Guerrier g;
  Voleur v;

  faire_rencontrer(g, v);
}
```

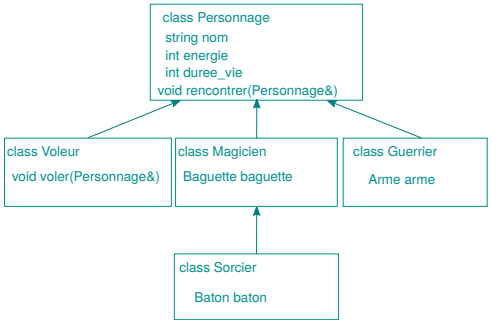
Polymorphisme (universel d'inclusion)

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**.




```
Personnage p1, p2;
// ...
p1.rencontrer(p2);
```

Résolution des liens

Une instance de sous-classe `B` est substituable à une instance de super-classe `A`.

Que se passe-t-il lorsque `B` redéfinit une méthode de `A` ?

```
class Personnage {
public:
  // ...
  void rencontrer(Personnage& p) const
  { cout << "Bonjour !" << endl; }
};


class Guerrier : public Personnage {
public:
  // ...
  void rencontrer(Personnage& p) const
  { cout << "Boum !" << endl; }
};
```

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

int main() {
  Guerrier g;
  Voleur v;

  faire_rencontrer(g, v);
}
```

Résolution des liens (2)

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

☞ résolution **statique** des liens

```
class Personnage {
public:
  // ...
  void rencontrer(Personnage& p) const
  { cout << "Bonjour !" << endl; }
};


class Guerrier : public Personnage {
public:
  // ...
  void rencontrer(Personnage& p) const
  { cout << "Boum !" << endl; }
};
```

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

int main() {
  Guerrier g;
  Voleur v;

  faire_rencontrer(g, v);
}
```

Résolution dynamique des liens

Il pourrait dans certains cas sembler plus naturel de choisir la méthode correspondant à la *nature réelle de l'instance*.

Dans ces cas, 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& autre) const
  { cout << "Bonjour !" << endl; }
};
```

Retour sur l'exemple (1)

```
class Personnage {
public:
  // ...
  virtual void rencontrer(
                    Personnage& p) const
  { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
  // ...
  void rencontrer(Personnage& p) const
  { cout << "Boum !" << endl; }
};
```

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

int main() {
  Guerrier g;
  Voleur v;

  faire_rencontrer(g, v);
}
```

... il **manque** encore un petit quelque chose pour que ça marche.

Retour sur l'exemple (2)

Attention ! Il faut passer `un` **par référence** pour que la fonction `faire_rencontrer` agisse sur l'instance d'origine !

```
class Personnage {
public:
  // ...
  virtual void rencontrer(
                    Personnage& p) const
  { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
  // ...
  void rencontrer(Personnage& p) const
  { cout << "Boum !" << endl; }
};
```

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

int main() {
  Guerrier g;
  Voleur v;

  faire_rencontrer(g, v);
}
```

Cette fois tout fonctionne comme on veut !

virtuelle / non virtuelle : un exemple

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

class Mammifere {
public:
           Mammifere() { cout << "Un nouveau mammifère est né !"          << endl; }
  virtual ~Mammifere() { cout << "Un mammifère est en train de mourir :(" << endl; }
          void manger()  const { cout << "Miam... croumf !"               << endl; }
  virtual void avancer() const { cout << "Un grand pas pour l'humanité."  << 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; }
};
```

virtuelle / non virtuelle : un exemple (2)

Que produit le code suivant ?

```
int main() {
  Mammifere* lui(new Dauphin());
  lui->avancer();
  lui->manger();
  delete lui;
  return 0;
}
```

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

```
Mammifere::Mammifere()
Dauphin::Dauphin()
Dauphin::avancer()
Mammifere::manger()
Dauphin::~Dauphin()
Mammifere::~Mammifere()
```

virtuelle / non virtuelle : un exemple (3)

Et si le destructeur de `Mammifere` n'avait pas été virtuel ?

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

Méthodes virtuelles : résumé et compléments

En résumé :
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
  (avancé ! Détails à suivre.)

Méthodes virtuelles et constructeurs

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

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

class A {
public:
  A() { f(); }
  virtual void f() const { cout << "A::f()" << endl; }
};

class B : public A {
public:
  virtual void f() const { cout << "B::f()" << endl; }
};

int main()
{
  A a;
  B b;
  A* pa(&b);
  pa->f();
  return 0;
}
```

Masquage, substitution et surcharge

Nous avons rencontré 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 (ou redéfinition, *overriding*), dans les sous-classes, de nouvelles versions de *méthodes virtuelles*.

Pour les *méthodes virtuelles*, on pourrait donc avoir les trois ? !

☞ **qui est quoi exactement ?**

Par ailleurs, C++11 introduit deux nouveaux mots clés, optionnels, pour justement aider le programmeur à préciser ses intentions :
`override` et `final`

Masquage, substitution et surcharge : définitions

- **surcharge** : même nom, mais paramètres différents, dans la même portée
  (Note : en C++, il ne peut y avoir surcharge que dans la même portée.
  Cf leçon à venir sur l'héritage multiple)
- **masquage** : entités de mêmes noms mais de portées différentes, masqués par les règles de résolution de portée.
  Pour les méthodes :
  - Attention aux subtilités : une seule méthode de *même nom* suffit à les masquer toutes, indépendemment des paramètres !
- **substitution**/redéfinition 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, substitution et surcharge : exemples

```
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)
  virtual void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};

class C : public A {
public:
 // introduction d'une 3e => masquage des 2 autres
  virtual void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
};
```

Masquage, substitution et surcharge : exemples

```
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;
}
```

Masquage, substitution et surcharge : exemples

```
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;
}
```

C++11 override et final

En C++11, le programmeur *peut* (optionnel) indiquer ses intentions lors de la (re)déclaration d'une méthode :

- avec le qualificatif `override` pour dire qu'il pense substituer/redéfinir une méthode virtuelle
- avec le qualificatif `final`, empêcher la substitution/redéfinition future d'une méthode virtuelle.

C++11 override et final : exemple

```
class A {
 // ...
 virtual void f1();
 virtual void f2() const;
         void f3();       // non virtuelle (oubli ?)
 virtual void f4() final; // pas de redéfinition
};

class B : public A {
 // ...
 virtual void f1() override; // OK
 virtual void fl() override; // Erreur faute de frappe : 1 <-> l
 virtual void f2() override; // Erreur: a oublié le const
         void f3() override; // Erreur: non virtuelle
 virtual void f4();          // Erreur : f4 était final
};
```

C++11 override et final

Conseils :

- `override` : utilisez-le (pour vous prémunir), même si c'est un peu verbeux
- `final` : oubliez (pour les méthodes)

Note : le mot clé `final` peut aussi s'utiliser pour les classes elles-mêmes pour empêcher la dérivation (interdire les sous-classes) :

```
class Sterile final { ... };
class C : public Sterile // INTERDIT !
```

Méthodes virtuelles pures : problème

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

Besoin de méthodes virtuelles pures : exemple

Exemple :

```
class FigureFermee  {
  //  ...

  // difficile à définir à ce niveau !..
  virtual double surface(...) const { ??? }

  // ...pourtant la méthode suivante en aurait besoin !
  double volume(double hauteur) const {
    return hauteur * surface();
  }
};
```

Définir `surface` de façon arbitraire sachant qu'on va la redéfinir plus tard n'est pas une bonne solution (source d'erreurs) !

☞ *Solution* : déclarer la méthode `surface` comme **virtuelle pure**

Besoin de méthodes virtuelles pures : autre exemple (1)

Plusieurs équipes collaborent à la création d'un jeu.

Une équipe prend en charge les classes de base suivantes :

- `Jeu` :
  - Classe pour gérer le jeu
  - Se contente ici d'afficher les personnages
- `Personnage` :
  - Classe de base pour les personnages

Une autre équipe ajoutera des sous-classes de personnages spécifiques.

Besoin de méthodes virtuelles pures : autre exemple (2)

La classe `Jeu` développée par la première équipe :

☞ gère un tableau de personnages et les affiche

```
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;
};
```

Besoin de méthodes virtuelles pures : autre exemple (3)

Si l'on ne met aucune méthode `afficher` dans `Personnage`, la classe `Jeu` ne compile pas :

```
class Jeu {
  // ...
  void afficher() const {
    for (auto un_perso : persos) {
      un_perso->afficher(); // ERREUR !
    }
  }
  // ...
};
```

☞ On **doit** donc mettre une méthode `afficher` dans la classe `Personnage`...

De plus, on aimerait :

- imposer aux sous-classes (`Guerrier`, ...) d'avoir leur méthode `afficher` spécifique
- que cette méthode spécifique à la sous-classe soit exécutée (polymorphisme ☞ *donc* méthode **virtuelle**)

Besoin de méthodes virtuelles pures : autre exemple (4)

☞ On **doit** donc mettre une méthode *virtuelle* `afficher` dans la classe `Personnage`...

...mais comment faire ?

```
class Personnage {
  // ...
  virtual void afficher() const {
    // Comment afficher un personnage générique ?
  }
  // ...
};
```

Et comment *imposer* que la méthode `afficher` soit redéfinie dans les sous-classes ?

Besoin de méthodes virtuelles pures : autre exemple (5)

Première « solution » :

ajouter une méthode quelconque définie arbitrairement :

```
class Personnage {
  // ...
  // On n'affiche rien : corps de la méthode vide
  virtual void afficher() const { }
  // ...
};
```

C'est une **très mauvaise idée**

☞ Mauvais modèle de la réalité
(affichage incorrect si une sous-classe ne redéfinit pas la méthode : personnages fantômes !)

☞ Cette solution n'impose pas que la méthode `afficher` soit redéfinie

Besoin de méthodes virtuelles pures : autre exemple (6)

Bonne solution :

Signaler que la méthode doit exister dans *chaque* sous-classe sans qu'il soit nécessaire de la définir dans la super-classe

☞ 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 {
  // ...
  virtual void afficher() const = 0;
  // ...
};
```

Méthodes virtuelles pures : autre exemple

```
class FigureFermee {
   public:
     virtual double surface()   const = 0;
     virtual double perimetre() const = 0;

  // On peut utiliser une méthode virtuelle pure :
  double volume (double hauteur) const {
    return hauteur * surface();
  }
};
```

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` »).)

Un exemple « concret »...

Classes abstraites : exemple

Une autre équipe crée la sous-classe `Guerrier` de `Personnage` et veut l'utiliser :

```
Jeu jeu;
jeu.ajouter_personnage(new Guerrier(...));
```

S'ils ont oublié de définir la méthode `afficher`,
le code ci-dessus génère une erreur de compilation
car on ne peut pas créer d'instance de `Guerrier` :

```
cannot allocate an object of abstract type ‘Guerrier’
because the following virtual functions are pure within ‘Guerrier’:
virtual void Guerrier::afficher()
```

Classes abstraites : autre exemple

```
class Cercle: public FigureFermee {
public:
  double surface() const override {
    return M_PI * rayon * rayon;
  }
  double perimetre() const override {
    return 2.0 * M_PI * rayon;
  }
protected:
  double rayon;
};
```

`Cercle` n'est pas une classe abstraite

```
class Polygone: public FigureFermee {
public:
  double perimetre() const override {
    double p(0.0);
    for (auto cote : cotes) {
      p += cote;
    }
    return p;
  }
protected:
  vector <double> cotes;
};
```

`Polygone` *reste* par contre une classe *abstraite*

Collection hétérogène

Nous avons vu jusqu'à maintenant que :

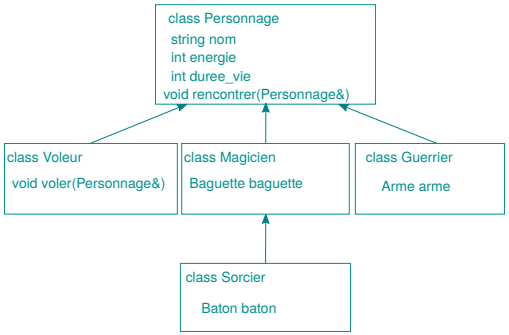
- l'*héritage* et les *méthodes virtuelles* permettent de mettre en œuvre des **traitements génériques** sur les instances d'une hiérarchie de classes (polymorphisme d'inclusion).
- les fonctions/méthodes génériques doivent utiliser des arguments **passés par référence** pour que le traitement se fasse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

Qu'en est-il si un tel traitement (générique) doit porter sur un *ensemble* d'instances d'une hiérarchie de classe ?

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

Collection hétérogène : exemple

Rappelez-vous de l'exemple du jeu avec des `Personnage`s :



Comment gérer les différentes classes de `Personnage`s dans le `Jeu` ?

Collection hétérogène : exemple (2)

On pourrait définir une classe `Jeu` comme suit :

```
class Jeu {
public:
  void afficher_guerriers() const;
  void afficher_magiciens() const;
  // ...
  void ajouter_guerrier(const Guerrier&);
  void ajouter_magicien(const Magicien&);
  // ...
private:
  vector<Guerrier> guerriers;
  vector<Magicien> magiciens;
};
```

C'est une solution possible (un point de vue), mais pas nécessairement la seule. On pourrait vouloir regrouper la gestion de tous les personnages et avoir ainsi une solution plus concise...

Collection hétérogène : exemple (3)

On pourrait par exemple souhaiter plutôt écrire quelque chose comme :

```
class Jeu {
public:
  void afficher() const;
  void ajouter_personnage(const Personnage&);

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

☞ les instances contenues dans l'attribut `personnages` font partie d'une *même hiérarchie de classe*, mais sont de nature *hétérogène* (`Guerrier`, `Magicien`, ...).

On pourrait par exemple vouloir que, si `personnages[i]` est un guerrier, la méthode `personnages[i].afficher()` soit bien celle de la sous-classe `Guerrier`.

Résolution dynamique des liens

Mais le code ci-contre **ne** permet **pas** le comportement polymorphique :

l'attribut `personnages` est constitué d'*instances* de type `Personnage` et non pas de *références/pointeurs* à ces instances.

```
class Jeu {
public:
  void afficher() const;
  void ajouter_personnage(const Personnage&);

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

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

Rappel sur les pointeurs/références :
«*Utilisez des références quand vous pouvez, des pointeurs quand vous devez.*»

Collection de pointeurs

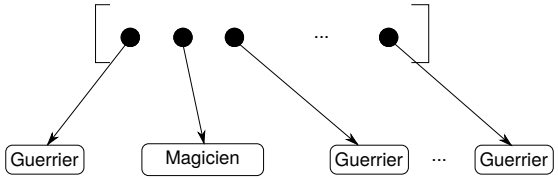
Malheusement on ne peut pas mettre de référence dans un `vector`.

La solution à ce problème consiste donc à passer par un vecteur de **pointeurs** :

```
class Jeu {
  //...
  vector<Personnage*> personnages;
};
```

```
#include <memory>
// ...
class Jeu {
  //...
  vector<unique_ptr<Personnage>> personnages;
};
```

Notez que donc 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 : classes

Comment l'utiliser ?

Le plus simple, comme dans la séquence vidéo précédente :

```
Jeu jeu;
jeu.ajouter_personnage(new Guerrier(...));
```

On aurait donc :

```
class Jeu {
public:
  void afficher() const;
  void ajouter_personnage(Personnage*);

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

Exemple complet : méthodes

```
void Jeu::ajouter_personnage(Personnage* nouveau) {
  if (nouveau != nullptr) {
    personnages.push_back(unique_ptr<Personnage>(nouveau));
  }
}
```

```
void Jeu::afficher() const {
  for (auto quidam : personnages) {
    quidam->afficher();
  }
}
```

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

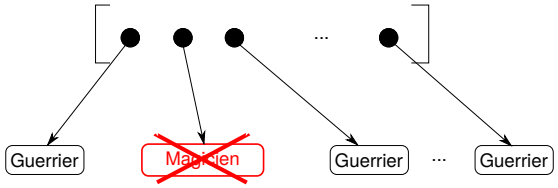
Exemple complet : utilisation

```
Jeu jeu;
jeu.ajouter_personnage(new Guerrier(...));
jeu.ajouter_personnage(new Magicien(...));
jeu.ajouter_personnage(new Voleur(...));
jeu.ajouter_personnage(new Guerrier(...));
// ...
jeu.afficher();
```

Pointeurs et intégrité des données

Cette classe `Jeu` comporte cependant *un danger potentiel* :

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



Attention ! La co-existence des pointeurs et des éléments pointés n'est cependant pas du tout garantie !

Au programmeur de ne pas faire de bêtises.

Exemple...

Pointeurs et intégrité des données : *mauvais* exemple

```
void creer_magicien(Jeu& jeu) {
  Magicien mago(...);
  jeu.ajouter_personnage(&mago);
}
// ...
int main() {
  Jeu mon_jeu;
  creer_magicien(mon_jeu);
  mon_jeu.afficher(); // ouille !
  return 0;
}
```

La fonction `creer_magicien` ajoute un nouveau magicien au jeu `mon_jeu`, mais par le biais d'une *variable locale* (bouh !)

Une fois l'exécution de `creer_magicien` terminée, la **variable locale est détruite** !

Attention ! Le pointeur stocké dans le vecteur `personnages` **existe toujours**...

Allocation/désallocation dynamique

La solution à ce problème est que l'utilisateur **alloue dynamiquement** une portion de mémoire *qui sera préservée* après la fin du bloc où l'on crée l'instance.

Exemple :

```
// définition robuste de la fonction creer_magicien
void creer_magicien(Jeu& jeu) {
  jeu.ajouter_personnage(new Magicien(...));
}
```

Grâce à l'utilisation du `new`, la **mémoire allouée dynamiquement** pour le magicien créé dans `creer_magicien` est préservée à la fin de l'exécution de cette fonction.

`unique_ptr` ?

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 !

Collection de pointeurs : lesquels ?

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

Par exemple ici avec des pointeurs « à la C » :

```
class Jeu {
public:
  void afficher() const;
  void ajouter_personnage(Personnage*);

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

Rappel d'utilisation :

```
Jeu jeu;
jeu.ajouter_personnage(new Guerrier(...));
```

Exemple, complet ?

```
class Jeu {
public:
  void afficher() const;
  void ajouter_personnage(Personnage*);

private:
  vector<Personnage*> personnages;
};

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

void Jeu::ajouter_personnage(Personnage* nouveau) {
  if (nouveau != nullptr) {
    personnages.push_back(nouveau);
  }
}
```

Exemple précédent (`unique_ptr`)

```
class Jeu {
public:
  void afficher() const;
  void ajouter_personnage(Personnage*);

private:
  vector<unique_ptr<Personnage>> personnages;
};

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

void Jeu::ajouter_personnage(Personnage* nouveau) {
  if (nouveau != nullptr) {
    personnages.push_back(unique_ptr<Personnage>(nouveau));
  }
}
```

Gare aux pointeurs !

Qui dit «pointeurs», dit aussi « bonne gestion » et « programmation rigoureuse »...

En particulier pensez, si nécessaire, à la *copie profonde* et au *destructeur* pour libérer la mémoire allouée

Et n'oubliez pas la règle d'or :
c'est celui qui a alloué la mémoire (`new`) **qui est chargé de la libérer** (`delete`)

Par exemple ici, fournir une fonction

```
void Jeu::detruire_personnage(Personnage* qui);
```

ou

```
void Jeu::detruire_personnage(size_t index);
```

ou alors (voire les deux) :

```
void Jeu::detruire_tout();
```

(Rappel : `jeu.ajouter_personnage(new Magicien(...));`)

Libération mémoire (1)

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

Libération mémoire (2)

```
void Jeu::detruire_personnage(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
    // supression plus couteuse qui préserve l'ordre
  personnages.erase(personnages.begin() + lequel);
}
```

Pointeurs « à la C »

Problème potentiel avec des pointeurs « à la C » :
intégrité des données

3 facettes :

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

Partage des données ?

Considérons un programme de modélisation graphique manipulant des *dessins*, lesquels sont des ensembles de *figures* géométriques :

- `Figure` comme classe abstraite, avec différentes sous-classes concrètes (cercles, rectangles, carrés, ...)
- `Dessin` comme *collection hétérogène* de figures.

Le contenu d'un `Dessin` est-il **personnel** ou **partagé** ?

Par exemple, si l'on colorie en rouge le cercle 23 du dessin 18, est-ce que seul ce cercle sera rouge ou bien d'autres ? du même dessin ? d'autres dessins ?

Les réponses à ces questions dépendent du cadre général du programme et de sa **conception**, et n'ont pas de réponse unique.

Partage des données ?

Le contenu **personnel** ou **partagé** ?

- Dans le cas des dessins, il semble naturel que les éléments de la collection (les figures) soient uniques et personnels.
- Dans le cas des personnages *du* jeu aussi (avec qui partager ?)

☞ Ceci est *garanti* par les `unique_prt` !