

Support de cours

Cours:

Introduction à la programmation orientée objet (en C++)

Vidéo:

W15-01-intropolym-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

Personnages de la hiérarchie. Types des données. Hiérarchie de classes. Personnage principal. Personnages de natures diverses. Ensemble d'autres personnages. Vecteur de personnage. Formes diverses. Variable perso. Types de données. Notions fondamentales. Tour de jeu. Façon générale. Instances d'une sous-classe. Ensemble des personnages.



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en C++).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>

Polymorphisme et résolution dynamique des liens

(Partie 1)

Introduction à la programmation orientée objet (en C++)

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



Concepts fondamentaux de l'orienté-objet :

- ▶ Encapsulation
- ▶ Abstraction
- ▶ Héritage
- ▶ Polymorphisme



Vous connaissez maintenant les notions fondamentales de l'orientée objet

notes

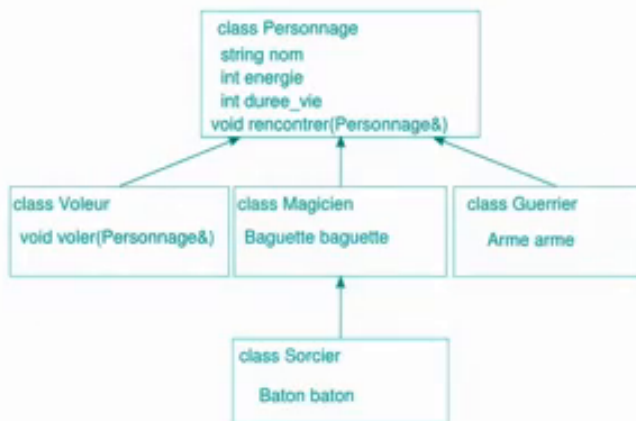
résumé

0m 1s



Polymorphisme ?

Un exemple :



que sont l'encapsulation, l'abstraction et l'héritage. Il est temps d'aborder une notion tout aussi centrale, celle du polymorphisme. De façon générale, le polymorphisme est le fait qu'un même code puisse s'adapter automatiquement aux types des données auxquels il s'applique. Il s'agit d'une notion importante car elle permet d'aller vers du code générique, du code qui s'écrit de façon unifiée pour différents types de données.

notes

résumé

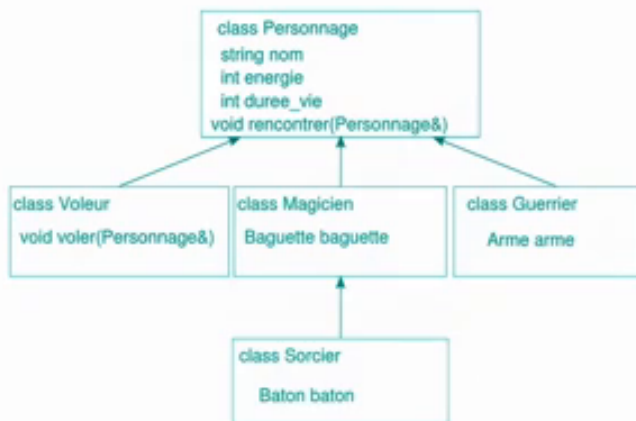
0m 5s



Polymorphisme ?

Personnage

Un exemple :



Pour introduire un peu plus concrètement la notion de polymorphisme, partons d'un exemple. Supposons que l'on souhaite programmer un jeu qui met en scène les personnages de la hiérarchie qu'on a déjà eu l'occasion d'évoquer dans les séquences précédentes. Supposons que notre jeu comporte un personnage principal qui serait le joueur.

notes

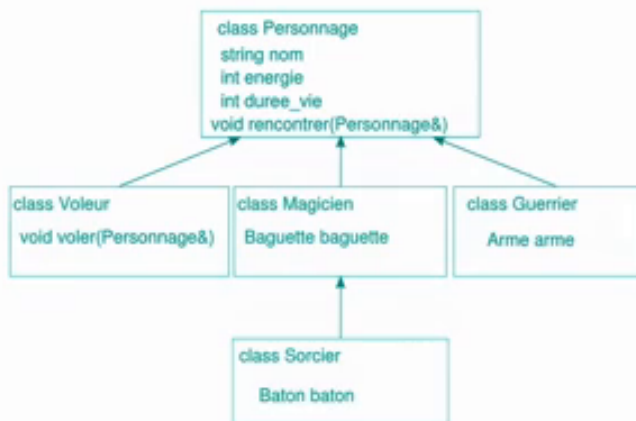
résumé

0m 30s



Polymorphisme ?

Un exemple :



Personnage joueur:
vector < Personnage > autres;

Et que dans un tour de jeu, ce dernier soit amené à faire des rencontres c'est-à-dire qu'il soit rencontré par tout un ensemble d'autres personnages. L'ensemble des personnages en C++ pourrait naturellement prendre l'allure suivante : un vecteur de Personnage,

notes

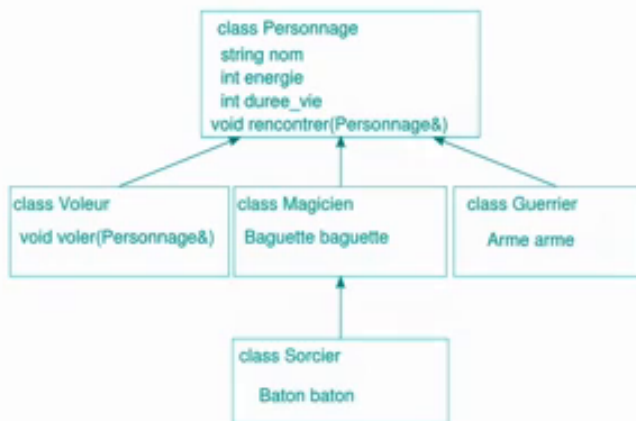
résumé

0m 49s



Polymorphisme ?

Un exemple :



Personnage joueur:
 vector< Personnage > autres;
 for (auto p

qui serait les autres et pour écrire le code qui fait que les autres rencontrent le joueur, une bouclefor* comme ceci, par exemple

notes

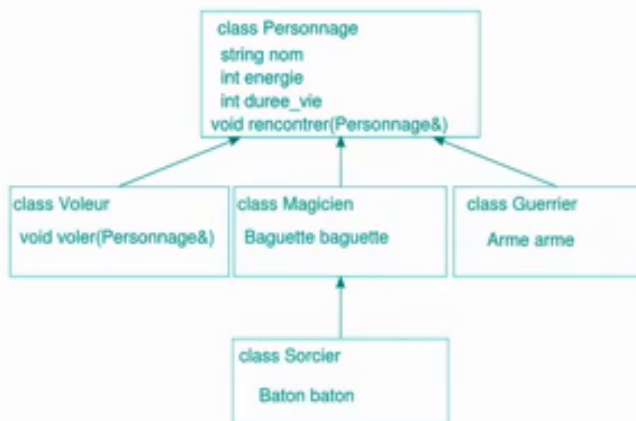
résumé

1m 13s



Polymorphisme ?

Un exemple :



Personnage joueur:
 vector< Personnage > autres;
 for (auto perso : -

où chacun des autres personnages rencontrent tour à tour notre joueur.

notes

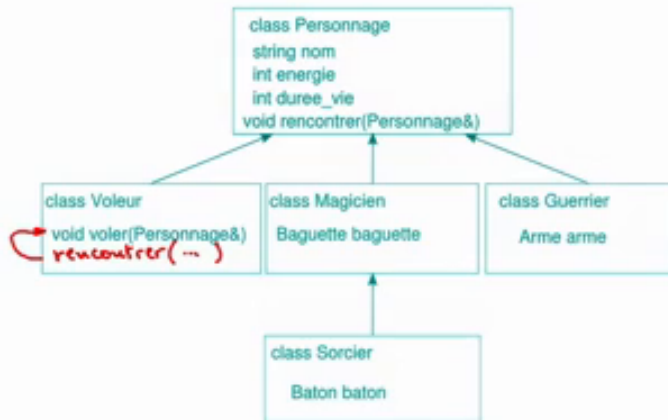
résumé

1m 25s



Polymorphisme ?

Un exemple :



Personnage joueur:

```
vector< Personnage > autres;
```

```
for (auto perso : autres) {
    perso.rencontrer(joueur);
}
```

Pour que notre jeu ne soit pas trop monotone, nous aimerions évidemment avoir dans la collection des autres personnages, des personnages de natures diverses, de formes diverses, c'est-à-dire qu'ici dans cette variable perso, nous aurions tour à tour un guerrier, un voleur, un sorcier. Nous savons aussi depuis une séquence précédente qu'une sous-classe peut avoir sa propre définition plus spécifique d'une méthode déjà définie plus haut. Par exemple, on pourrait imaginer que lorsque le Voleur rencontre le Personnage, il le vole.

notes

résumé

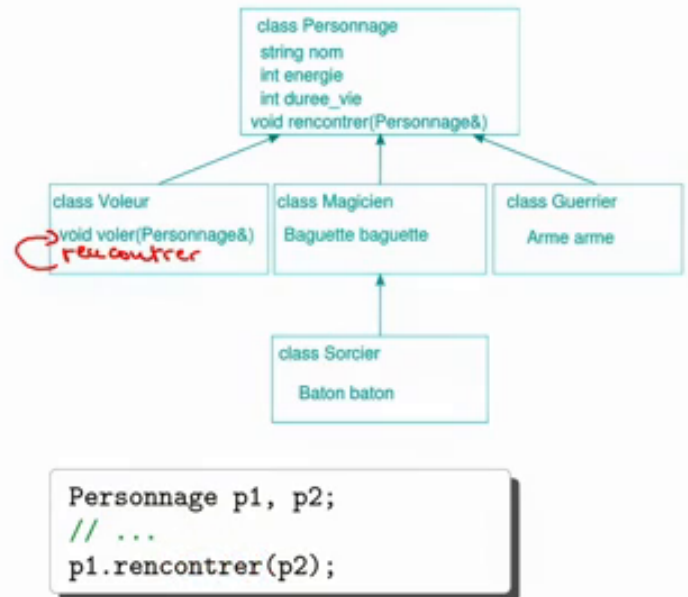
1m 27s



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.**



Le polymorphisme est le fait d'appliquer le même code à des objets prenant plusieurs formes et que l'exécution du code s'adapte automatiquement à la forme de l'objet. Cela veut dire concrètement que si à un moment donné, notre personnage est un voleur, la méthode rencontrer va être choisie de sorte à être celle du voleur qui va donc trousseur notre personnage principal sans autre forme de procès. L'idée est que grâce à l'héritage, le même code peut s'appliquer à toutes les formes de personnages possibles. Toutes ces formes de personnages ont une méthode rencontrer que la façon dont un personnage en rencontre un autre peut prendre différentes formes, c'est-à-dire concrètement qu'un magicien peut saluer un Personnage, un guerrier peut le frapper, un voleur peut le voler et ainsi de suite. Et grâce au polymorphisme, le même code appliqué à différents personnages pourra avoir un comportement différent propre à chacun. Très concrètement, si je mets un Voleur dans un Personnage, dans une solution polymorphique,

notes

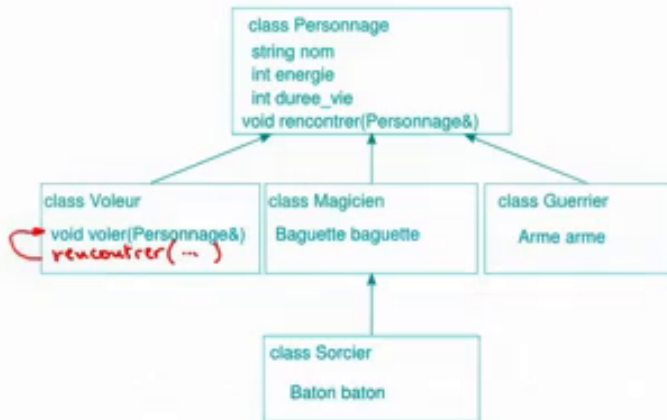
résumé

2m 25s



Polymorphisme ?

Un exemple :



```

Personnage joueur;
vector< Personnage > autres;

for (auto perso : autres) {
    perso.rencontrer(joueur);
}
  
```

la méthode rencontrer s'adapterait automatiquement à la nature réelle de l'objet contenu dans la variable Personnage et donc ici, on appliquerait la méthode rencontrer du voleur. L'intérêt de manipuler un voleur sous la forme d'un personnage et non pas directement sous la forme d'un voleur est justement de pouvoir faire des manipulations unifiées sur toutes sortes de types d'objets.

notes

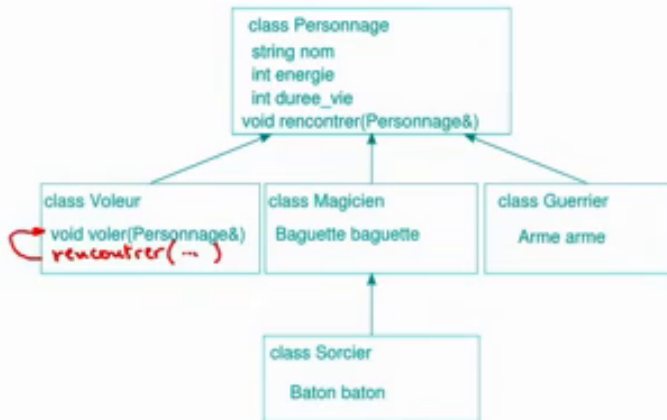
résumé

3m 37s



Polymorphisme ?

Un exemple :



Personnage joueur:

```
vector< Personnage > autres;
```

```
for (auto perso : autres) {
    perso.rencontrer(joueur);
}
```

```
Guerrier g(...);
Sorcier s(...);
autres[0] = g;
autres[1] = s;
...
```

Vous en avez un exemple concret ici où tout un ensemble de personnages de natures diverses sont manipulés de façon unifiée en étant placé dans un tableau de `Personnage`. Ceci présuppose qu'à un moment donné, j'ai dû mettre dans ma collection de personnages des personnages de nature diverse, c'est-à-dire que j'ai dû faire des manipulations qui ressemblent à ceci. Par exemple, si j'ai un objet de type `Guerrier` et un objet de type `Sorcier` alors, j'ai rempli mon tableau de personnages en faisant par exemple quelque chose comme ceci. C'est-à-dire, puisque les entrées de mon tableau sont de type `Personnage` que l'on a mis dans un `Personnage` un objet de type `Guerrier`.

notes

résumé

3m 56s



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;
```

Petit rappel sur l'héritage, affecter un Guerrier à un Personnage

notes

résumé

4m 49s



« 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);
}
```

est tout à fait licite. Dans une hiérarchie de classes, la sous-classe hérite de la super-classe, non seulement, les attributs/les méthodes mais également le type. Un Guerrier est aussi un personnage et de ce fait, cette affectation est tout à fait licite. Rappelons également que l'héritage est transitif c'est-à-dire que je pourrai tout autant affecter un Sorcier à un Personnage. Pourquoi ? Parce que l'héritage étant transitif, un Sorcier est un Magicien qui est un Personnage à son tour. Le Sorcier est aussi un Personnage. On a donc le droit de faire correspondre à un objet de type super-classe un objet de type sous-classe. On en voit un exemple ici dans le cadre de l'affectation. Ce droit existe également dans le cadre du passage des paramètres et nous allons en voir un exemple sur le transparent suivant. On suppose ici l'existence d'une hiérarchie de classes telle que décrite dans l'exemple précédent. Un programmeur qui utilise cette hiérarchie de classe à programmé une fonction faire_rencontrer qui permet à un personnage d'en rencontrer un autre.

notes

résumé

4m 50s



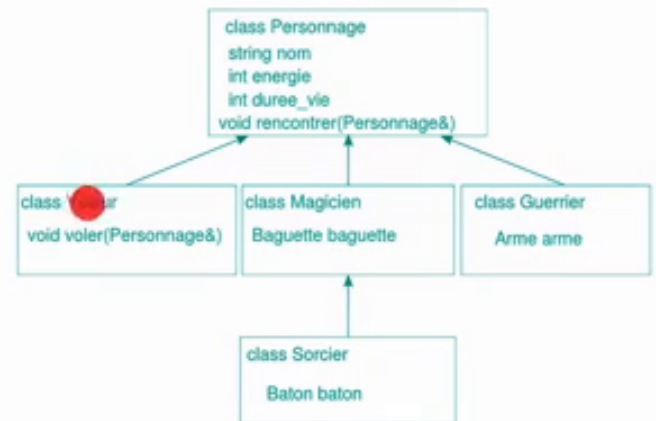
Polymorphisme (universel d'inclusion)

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);
  
```

Cette fonction faire_rencontrer est utilisée dans son programme principal et il lui est passée en argument un premier personnage qui est un guerrier et un second personnage qui est un voleur. Tout ceci pour montrer qu'on peut tout à fait mettre un Guerrier dans un Personnage ou un Voleur dans le cadre du passage des paramètres. Mettre un objet de type sous-classe dans une variable de type super-classe est autorisé aussi bien dans le cadre de l'affectation que celui du passage des paramètres. Ceci grâce à l'héritage du type dans une hiérarchie, un Guerrier est un Personnage. Un peu plus formellement donc le polymorphisme est le fait que des instances d'une sous-classe soient substituables aux instances des classes de leur ascendance, des super-classes soit en arguments de méthode soit lors d'affectation ce que l'on vient de voir dans les transparents précédents tout en gardant leurs propriétés propres. Dans le cas de cet exemple, l'idée serait donc de mettre un Voleur dans un personnage tout en garantissant que l'exécution de la méthode rencontrer s'adaptent automatiquement à la nature réelle de l'instance contenue dans la variable Personnage. C'est-à-dire qu'on appliquerait la méthode rencontrer du voleur.

notes

résumé

6m 1s



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);
}
```

Le fait de pouvoir mettre un Voleur dans un Personnage est une conséquence directe de l'héritage. Par contre, le fait que la méthode à invoquer soit choisie pendant l'exécution en fonction de la nature réelle des instances dépend de ce qu'on appelle la résolution dynamique des liens. Voyons maintenant de quoi il s'agit concrètement. Revenons pour cela à notre exemple précédent où nous avons substitué un objet de type sous-classe à une instance de type super-classe ici. Nous avons mis un Guerrier dans un Personnage qui est le paramètre un de la méthode faire_rencontrer et nous appelons la méthode rencontrer sur le Personnage en question. Nous nous plaçons ici dans le contexte où notre hiérarchie de classe aurait deux méthodes rencontrer. Une définie dans la super-classe qui est la méthode par défaut, la méthode générale et une spécialisée pour les guerriers dans la sous-classe Guerrier. Laquelle des deux méthodes rencontrer va ici être appelée ? En y regardant de plus près, on se rend compte qu'il existe deux points de vue possible. Le premier consiste à dire que le paramètre est de type Personnage, il faudra aller chercher la méthode rencontrer dans la classe Personnage. Le second qui consiste à dire on a mis dans Personnage un Guerrier et c'est dans la classe Guerrier qu'il faut aller chercher la méthode rencontrer. Il y a donc un point de vue qui consiste à dire que c'est le type de la variable qui prime pour le choix de la méthode et un second point de vue qui consiste à dire c'est le type de l'objet effectivement contenu dans la variable qui va être décisif pour le choix de la méthode. En C++, si l'on ne fait rien de particulier, c'est le type de la variable qui détermine la méthode à exécuter. On a ce qu'on appelle la résolution statique des liens ce qui

notes

résumé

7m 13s



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);
}
```

veut dire qu'ici c'est la méthode rencontrer de Personnage qui va être appliquée. On parle de résolution statique parce que le choix de la méthode peut se faire statiquement avant même que le programme ne s'exécute pendant la compilation. Dans cette phase là, on peut déjà savoir de quel type est une variable et sur la base de ce type, choisir la méthode à appliquer. choisir la méthode à appliquer.

notes

résumé