

# Programmez avec le langage C++

50 heures  Difficile

Mis à jour le 06/02/2020



## Créez les classes (Partie 1/2)

 [Connectez-vous](#) ou [inscrivez-vous](#) gratuitement pour bénéficier de toutes les fonctionnalités de ce cours !

Au chapitre précédent, vous avez vu que la programmation orientée objet pouvait nous simplifier la vie en « masquant », en quelque sorte, le code complexe. C'est un des avantages de la POO mais ce n'est pas le seul, comme vous allez le découvrir petit à petit : les objets sont aussi facilement réutilisables et modifiables.

À partir de maintenant, nous allons apprendre à créer des objets. Vous allez voir que c'est tout un art et que cela demande de la pratique. Il y a beaucoup de programmeurs qui prétendent faire de la POO et qui le font pourtant très mal. En effet, on peut créer un objet de 100 façons différentes et c'est à nous de choisir à chaque fois la meilleure, la plus adaptée. Ce n'est pas évident, il faut donc bien réfléchir avant de se lancer dans le code comme des forcenés.

Allez, on prend une grande inspiration et on plonge ensemble dans l'océan de la POO !

### Créez une classe



Commençons par la question qui doit vous brûler les lèvres.

Je croyais qu'on allait apprendre à créer des objets, pourquoi tu nous parles de créer une classe maintenant ?

Quel est le rapport ?

Eh bien justement, pour créer un objet, il faut d'abord créer une classe !

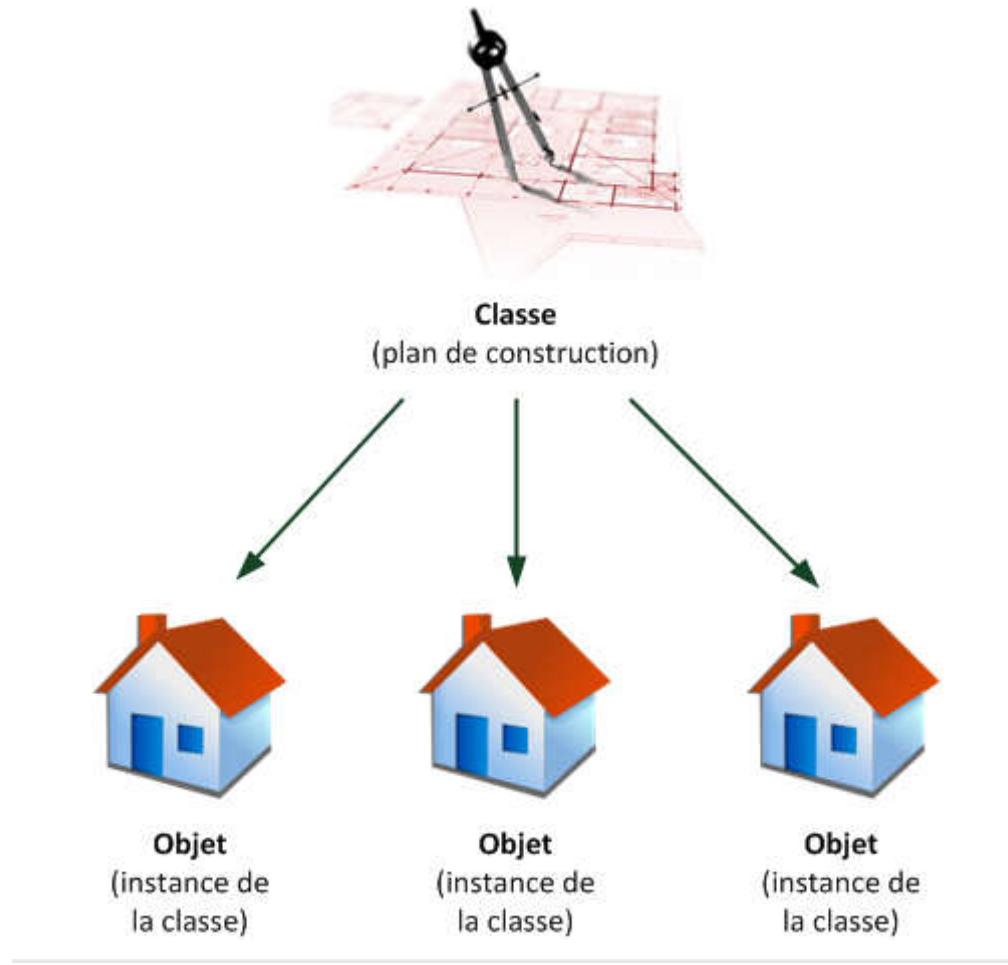
Je m'explique : pour construire une maison, vous avez besoin d'un plan d'architecte non ? Eh bien

Imaginez simplement que la classe c'est le plan et que l'objet c'est la maison.

« Créez une classe », c'est donc dessiner les plans de l'objet.

Une fois que vous avez les plans, vous pouvez faire autant de maisons que vous voulez en vous basant sur ces plans. Pour les objets c'est pareil : une fois que vous avez fait la classe (le plan), vous pouvez créer autant d'objets du même type que vous voulez.

Vocabulaire : on dit qu'un objet est une **instance** d'une classe. C'est un mot très courant que l'on rencontre souvent en POO. Cela signifie qu'un objet est la matérialisation concrète d'une classe (tout comme la maison est la matérialisation concrète du plan de la maison). Oui, je sais, c'est très métaphysique la POO mais vous allez voir, on s'y fait.



Une fois la classe créée, on peut construire des objets

## Créez une classe, oui mais laquelle ?

Avant tout, il va falloir choisir la classe sur laquelle nous allons travailler.

Reprenons l'exemple sur l'architecture : allons-nous créer un appartement, une villa avec piscine, un loft spacieux ?

En clair, quel type d'objet voulons-nous être capables de créer ?

Les choix ne manquent pas. Je sais que, quand on débute, on a du mal à imaginer ce qui peut être considéré comme un objet. La réponse est : presque tout !

Vous allez voir, vous allez petit à petit avoir le *feeling* qu'il faut avec la POO. Puisque vous débutez, c'est moi qui vais choisir (vous n'avez pas trop le choix, de toute façon !).

Pour notre exemple, nous allons créer une classe Personnage qui va représenter un personnage de jeu de rôle (RPG).

Si vous n'avez pas l'habitude des jeux de rôle, rassurez-vous : moi non plus. Pour suivre ce chapitre, vous n'avez pas besoin de savoir jouer à des RPG. J'ai choisi cet exemple car il me paraît didactique, amusant, et qu'il peut déboucher sur la création d'un jeu à la fin. Mais ce sera à vous de le terminer.

## Bon, on la crée cette classe ?

C'est parti.

Pour commencer, je vous rappelle qu'une classe est constituée (n'oubliez pas ce vocabulaire, il est fon-da-men-tal !) :

- de variables, ici appelées **attributs** (on parle aussi de **variables membres**) ;
- de fonctions, ici appelées **méthodes** (on parle aussi de **fonctions membres**).

Voici le code minimal pour créer une classe :

cpp

```
1 class Personnage
2 {
3
4 }; // N'oubliez pas le point-virgule à la fin !
```

Comme vous le voyez, on utilise le mot-clé **class**.

Il est suivi du nom de la classe que l'on veut créer. Ici, c'est Personnage .

Souvenez-vous de cette règle très importante : il faut que le nom de vos classes commence toujours par une lettre majuscule ! Bien que ce ne soit pas obligatoire (le compilateur ne hurlera pas si vous commencez par une minuscule), cela vous sera très utile par la suite pour différencier les noms des classes des noms des objets.

Nous allons écrire toute la définition de la classe entre les accolades. Tout ou presque se passera donc à l'intérieur de ces accolades.

Et surtout, très important, le truc qu'on oublie au moins une fois dans sa vie : *il y a un point-virgule après l'accolade fermante !*

## Ajout de méthodes et d'attributs

Bon, c'est bien beau mais notre classe **Personnage** est plutôt... vide.

Que va-t-on mettre dans la classe ? Vous le savez déjà voyons.

- des **attributs** : c'est le nom que l'on donne aux **variables** contenues dans des classes ;
- des **méthodes** : c'est le nom que l'on donne aux **fonctions** contenues dans des classes.

Le but du jeu, maintenant, c'est justement d'arriver à faire la liste de tout ce qu'on veut mettre dans notre **Personnage**. De quels attributs et de quelles méthodes a-t-il besoin ? C'est justement l'étape de *réflexion*, la plus importante. C'est pour cela que je vous ai dit au début de ce chapitre qu'il ne fallait surtout pas coder comme des barbares dès le début mais prendre le temps de *réfléchir*.

Cette étape de réflexion avant le codage est essentielle quand on fait de la POO. Beaucoup de gens, dont moi, ont l'habitude de sortir une feuille de papier et un crayon pour établir la liste des attributs et méthodes dont ils vont avoir besoin.

Un langage spécial, appelé **UML**, a d'ailleurs été spécialement créé pour concevoir les classes avant de commencer à les coder.

Par quoi commencer : les attributs ou les méthodes ? Il n'y a pas d'ordre, en fait, mais je trouve un peu plus logique de commencer par voir les attributs *puis* les méthodes.

## Les attributs

C'est ce qui va caractériser votre classe, ici le personnage. Ce sont des variables, elles peuvent donc évoluer au fil du temps. Mais qu'est-ce qui caractérise un personnage de jeu de rôle ? Allons, un petit effort.

- Par exemple, tout personnage a un niveau de vie. Hop, cela fait un premier attribut : **vie** ! On dira que ce sera un **int** et qu'il sera compris entre 0 et 100 (0 = mort, 100 = toute la vie).
- Dans un jeu de rôle, il y a le niveau de magie, aussi appelé **mana**. Là encore, on va dire que c'est un **int** compris entre 0 et 100. Si le personnage a 0 de mana, il ne peut plus lancer de sort magique et doit attendre que son mana se recharge tout seul au fil du temps (ou boire une potion de mana !).
- On pourrait rajouter aussi le nom de l'arme que porte le joueur : **nomArme**. On va utiliser pour cela un **string**.
- Enfin, il me semble indispensable d'ajouter un attribut **degatsArme**, un **int** qui indiquerait cette fois le degré de dégâts que porte notre arme à chaque coup.

On peut donc déjà commencer à compléter la classe avec ces premiers attributs :

cpp

```

1 class Personnage
2 {
3     int m_vie;
4     int m_mana;
5     string m_nomArme;
```

```

6     int m_degatsArme;
7 }

```

Deux ou trois petites choses à savoir sur ce code :

- Ce n'est pas une obligation mais une grande partie des programmeurs (dont moi) a l'habitude de faire commencer tous les noms des attributs de classe par « `m_` » (le « `m` » signifiant « membre », pour indiquer que c'est une variable membre, c'est-à-dire un attribut). Cela permet de bien différencier les attributs des variables « classiques » (contenues dans des fonctions par exemple).
- Il est impossible d'initialiser les attributs ici. Cela doit être fait via ce qu'on appelle un constructeur, comme on le verra un peu plus loin.
- Comme on utilise un objet `string`, il faut bien penser à rajouter un `#include <string>` dans votre fichier.

La chose essentielle à retenir ici, c'est que l'on utilise des attributs pour représenter la notion d'appartenance. On dit qu'un `Personnage` a une vie et a un niveau de magie. Il possède également une arme. Lorsque vous repérez une relation d'appartenance, il y a de fortes chances qu'un attribut soit la solution à adopter.

## Les méthodes

Les méthodes, elles, sont *grosso modo* les actions que le personnage peut effectuer ou qu'on peut lui faire faire. Les méthodes lisent et modifient les attributs.

Voici quelques actions réalisables avec notre personnage :

- `recevoirDegats` : le personnage prend un certain nombre de dégâts et donc perd de la vie.
- `attaquer` : le personnage attaque un autre personnage avec son arme. Il inflige autant de dégâts que son arme le lui permet (c'est-à-dire `degatsArme`).
- `boirePotionDeVie` : le personnage boit une potion de vie et regagne un certain nombre de points de vie.
- `changerArme` : le personnage récupère une nouvelle arme plus puissante. On change le nom de l'arme et les dégâts qui vont avec.
- `estVivant` : renvoie `true` si le personnage est toujours vivant (il possède plus que 0 point de vie), sinon renvoie `false`.

C'est un bon début, je trouve.

On va rajouter cela dans la classe avant les attributs (en POO, on préfère présenter les méthodes avant les attributs, bien que cela ne soit pas obligatoire) :

cpp

```

1 class Personnage
2 {
3     // Méthodes

```

```
4     void recevoirDegats(int nbDegats)
5     {
6     }
7     }
8
9     void attaquer(Personnage &cible)
10    {
11    }
12    }
13
14    void boirePotionDeVie(int quantitePotion)
15    {
16    }
17    }
18
19    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
20    {
21    }
22    }
23
24    bool estVivant()
25    {
26    }
27    }
28
29    // Attributs
30    int m_vie;
31    int m_mana;
32    string m_nomArme;
33    int m_degatsArme;
34 };
```

Je n'ai volontairement pas écrit le code des méthodes, on le fera après.

Ceci dit, vous devriez déjà avoir une petite idée de ce que vous allez mettre dans ces méthodes.

Par exemple, `recevoirDegats` retranchera le nombre de dégâts (indiqués en paramètre par `nbDegats`) à la vie du personnage.

La méthode `attaquer` est également intéressante : elle prend en paramètre... un autre personnage, plus exactement une référence vers le personnage cible que l'on doit attaquer ! Et que fera cette méthode, à votre avis ? Eh oui, elle appellera la méthode `recevoirDegats` de la cible pour lui infliger des dégâts.

Vous commencez à comprendre un peu comme tout cela est lié et terriblement logique ? On met en général un peu de temps avant de correctement « penser objet ». Si vous vous dites que vous n'auriez pas pu inventer un truc comme cela tout seul, assurez-vous : tous les débutants passent par là. À force de pratiquer, cela va venir.

Pour info, cette classe ne comporte pas toutes les méthodes que l'on pourrait y créer : par exemple, on n'utilise pas de magie ici. Le personnage attaque seulement avec une arme (une épée par

exemple) et n'emploie donc pas de sort. Je laisse exprès quelques fonctions manquantes pour vous inciter à compléter la classe avec vos idées.

En résumé, un objet est bel et bien un mix de variables (les attributs) et de fonctions (les méthodes). La plupart du temps, les méthodes lisent et modifient les attributs de l'objet pour le faire évoluer. Un objet est au final un petit système intelligent et autonome, capable de surveiller tout seul son bon fonctionnement.

## Droits d'accès et encapsulation



Nous allons maintenant nous intéresser au concept le plus *fondamental* de la POO :

**l'encapsulation.** Ne vous laissez pas effrayer par ce mot, vous allez vite comprendre ce que cela signifie.

Tout d'abord, un petit rappel. En POO, il y a deux parties bien distinctes :

- On **crée** des classes pour définir le fonctionnement des objets. C'est ce qu'on apprend à faire ici.
- On **utilise** des objets. C'est ce qu'on a appris à faire au chapitre précédent.

Il faut bien distinguer ces deux parties car cela devient ici très important.

### Création de la classe :

cpp

```
1 class Personnage
2 {
3     // Méthodes
4     void recevoirDegats(int nbDegats)
5     {
6
7     }
8
9     void attaquer(Personnage &cible)
10    {
11
12    }
13
14    void boirePotionDeVie(int quantitePotion)
15    {
16
17    }
18
19    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
20    {
21
22    }
23
24    bool estVivant()
25    {
26
27    }
```

```

29 // Attributs
30 int m_vie;
31 int m_mana;
32 string m_nomArme;
33 int m_degatsArme;
34 };

```

## Utilisation de l'objet :

cpp

```

1 int main()
2 {
3     Personnage david, goliath;
4     //Création de 2 objets de type Personnage : david et goliath
5
6     goliath.atttaquer(david); //goliath attaque david
7     david.boirePotionDeVie(20); //david récupère 20 de vie en buvant une potion
8     goliath.atttaquer(david); //goliath réattaque david
9     david.atttaquer(goliath); //david contre-attaque... c'est assez clair non ?
10
11    goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);
12    goliath.atttaquer(david);
13
14
15    return 0;
16 }

```

Tenez, pourquoi n'essaierait-on pas ce code ?

Allez, on met tout dans un même fichier (en prenant soin de définir la classe *avant* le `main()`) et zou !

cpp

```

1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 class Personnage
7 {
8     // Méthodes
9     void recevoirDegats(int nbDegats)
10    {
11    }
12
13
14     void attaquer(Personnage &cible)
15    {
16    }
17
18
19     void boirePotionDeVie(int quantitePotion)
20    {
21    }
22
23
24     void changerArme(string nomNouvelleArme, int degatsNouvelleArme)

```

```
25  {
26
27  }
28
29  bool estVivant()
30  {
31  }
32 }
33
34 // Attributs
35 int m_vie;
36 int m_mana;
37 string m_nomArme;
38 int m_degatsArme;
39 };
40
41 int main()
42 {
43     Personnage david, goliath;
44     //Création de 2 objets de type Personnage : david et goliath
45
46     goliath.atttaquer(david);    //goliath attaque david
47     david.boirePotionDeVie(20); //david récupère 20 de vie en buvant une potion
48     goliath.atttaquer(david);    //goliath réattaque david
49     david.atttaquer(goliath);    //david contre-attaque... c'est assez clair non ?
50
51     goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);
52     goliath.atttaquer(david);
53
54
55     return 0;
56 }
```

Compilez et admirez... la belle erreur de compilation !

```
Error : void Personnage::atttaquer(Personnage&) is private within this context
```

Encore une insulte de la part du compilateur !

## Les droits d'accès

On en arrive justement au problème qui nous intéresse : celui des droits d'accès (oui, j'ai fait exprès de provoquer cette erreur de compilation ; vous ne pensiez tout de même pas que ce n'était pas prévu ? ).

Ouvrez grand vos oreilles : chaque attribut et chaque méthode d'une classe peut posséder son propre droit d'accès. Il existe *grossost modo* deux droits d'accès différents :

- **public** : l'attribut ou la méthode peut être appelé depuis l'extérieur de l'objet.
- **private** : l'attribut ou la méthode ne peut pas être appelé depuis l'extérieur de l'objet. *Par défaut, tous les éléments d'une classe sont private*.

Il existe d'autres droits d'accès mais ils sont un peu plus complexes. Nous les verrons plus tard.

Concrètement, qu'est-ce que cela signifie ? Qu'est-ce que « l'extérieur » de l'objet ?

Eh bien, dans notre exemple, « l'extérieur » c'est le `main()`. En effet, c'est là où on utilise l'objet. On fait appel à des méthodes mais, comme elles sont par défaut privées, on ne peut pas les appeler depuis le `main()` !

Pour modifier les droits d'accès et mettre par exemple `public`, il faut taper « `public` » suivi du symbole « `:` » (deux points). Tout ce qui se trouvera à la suite sera `public`.

Voici ce que je vous propose de faire : on va mettre en `public` toutes les méthodes et en `privé` tous les attributs.

Cela nous donne :

cpp

```
1 class Personnage
2 {
3     // Tout ce qui suit est public (accessible depuis l'extérieur)
4     public:
5
6     void recevoirDegats(int nbDegats)
7     {
8
9     }
10
11    void attaquer(Personnage &cible)
12    {
13
14    }
15
16    void boirePotionDeVie(int quantitePotion)
17    {
18
19    }
20
21    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
22    {
23
24    }
25
26    bool estVivant()
27    {
28
29    }
30
31    // Tout ce qui suit est privé (inaccessible depuis l'extérieur)
32    private:
33
34    int m_vie;
35    int m_mana;
36    string m_nomArme;
37    int m_degatsArme;
```

38 };

Tout ce qui suit le mot-clé `public` : est public donc toutes nos méthodes sont publiques. Ensuite vient le mot-clé `private` : . Tout ce qui suit ce mot-clé est privé donc tous nos attributs sont privés.

Voilà, vous pouvez maintenant compiler ce code et vous verrez qu'il n'y a pas de problème (même si le code ne fait rien pour l'instant). On appelle des méthodes depuis le `main()` : comme elles sont publiques, on a le droit de le faire.

En revanche, nos attributs sont privés, ce qui veut dire qu'on n'a pas le droit de les modifier depuis le `main()` . En clair, *on ne peut pas écrire* dans le `main()` :

cpp

```
1 goliath.m_vie = 90;
```

Essayez, vous verrez que le compilateur vous ressort la même erreur que tout à l'heure : « ton bidule est private... bla bla bla... pas le droit d'appeler un élément private depuis l'extérieur de la classe ».

Mais alors... cela veut dire qu'on ne peut pas modifier la vie du personnage depuis le `main()` ? Eh oui ! C'est ce qu'on appelle **l'encapsulation**.

## L'encapsulation

Moi j'ai une solution ! Si on mettait tout en public ? Les méthodes et les attributs, comme cela on peut tout modifier depuis le `main()` et plus aucun problème ! Non ? Quoi j'ai dit une bêtise ?

Oh, trois fois rien. Vous venez juste de vous faire autant d'ennemis qu'il y a de programmeurs qui font de la POO dans le monde.

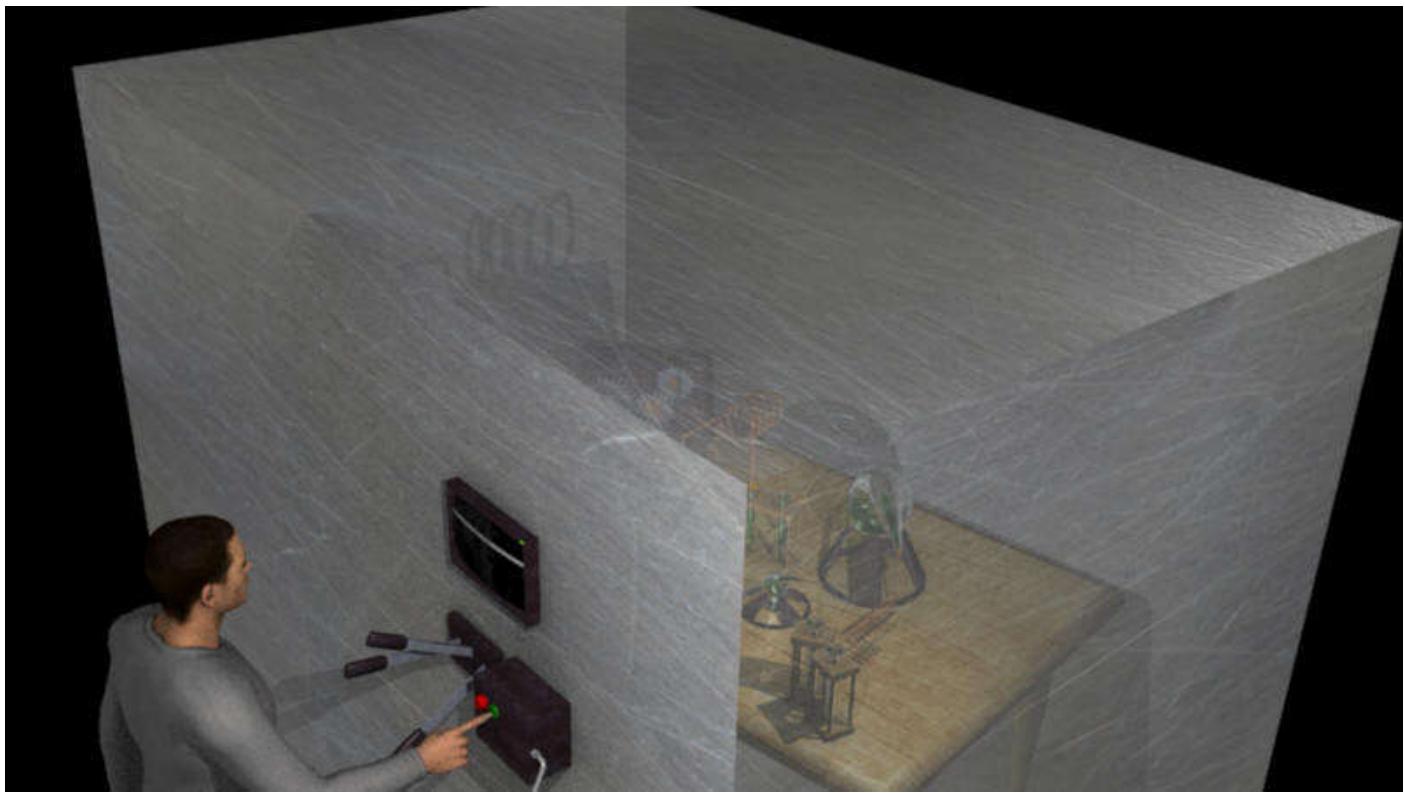
Il y a une règle d'or en POO et *tout* découle de là. S'il vous plaît, imprimez ceci en gros sur une feuille et placardez cette feuille sur un mur de votre chambre :

**Encapsulation : tous les attributs d'une classe doivent toujours être privés.**

Cela a l'air bête, stupide, irréfléchi, et pourtant tout ce qui fait que la POO est un principe puissant vient de là. Je ne veux pas en voir un seul mettre un attribut en `public` !

Voilà qui explique pourquoi j'ai fait exprès, dès le début, de mettre les attributs en privé. Ainsi, on ne peut pas les modifier depuis l'extérieur de la classe et cela respecte le principe d'encapsulation.

Vous vous souvenez de ce schéma du chapitre précédent ?



L'utilisation du code est simplifiée grâce à l'utilisation d'un objet

Les fioles chimiques, ce sont les **attributs**.

Les boutons sur la façade avant, ce sont les **méthodes**.

Et là, pif paf pouf, vous devriez avoir tout compris d'un coup. En effet, le but du modèle objet est justement de masquer à l'utilisateur les informations complexes (les attributs) pour éviter qu'il ne fasse des bêtises avec.

Imaginez par exemple que l'utilisateur puisse modifier la vie... qu'est-ce qui l'empêcherait de mettre 150 de vie alors que la limite maximale est 100 ? C'est pour cela qu'il faut *toujours* passer par des méthodes (des fonctions) qui vont *d'abord* vérifier qu'on fait les choses correctement avant de modifier les attributs.

Cela garantit que le contenu de l'objet reste une « boîte noire ». On ne sait pas comment cela fonctionne à l'intérieur quand on l'utilise et c'est très bien ainsi. C'est une sécurité, cela permet d'éviter de faire péter tout le bazar à l'intérieur.

Si vous avez fait du C, vous connaissez le mot-clé `struct`. On peut aussi l'utiliser en C++ pour créer des classes.

La seule différence avec le mot-clé `class` est que, par défaut, les méthodes et attributs sont publics au lieu de privés.

## Séparez prototypes et définitions



Bon, on avance mais on n'a pas fini !

Voici ce que je voudrais qu'on fasse :

- séparer les méthodes en prototypes et définitions dans deux fichiers différents, pour avoir un code plus modulaire ;
- implémenter les méthodes de la classe Personnage (c'est-à-dire écrire le code à l'intérieur parce que, pour le moment, il n'y a rien).

À ce stade, notre classe figure dans le fichier main.cpp , juste au-dessus du main() . Et les méthodes sont directement écrites dans la définition de la classe. Cela fonctionne, mais c'est un peu bourrin.

Pour améliorer cela, il faut tout d'abord clairement séparer le main() (qui se trouve dans main.cpp ) des classes.

Pour chaque classe, on va créer :

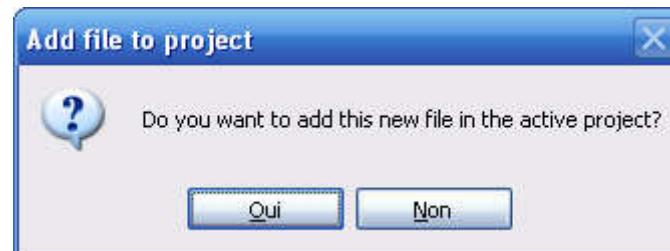
- un header (fichier \*.h ) qui contiendra les attributs et les prototypes de la classe ;
- un fichier source (fichier \*.cpp ) qui contiendra la définition des méthodes et leur implémentation.

Je vous propose d'ajouter à votre projet deux fichiers nommés très exactement :

- Personnage.h ;
- Personnage.cpp .

Vous noterez que je mets aussi une majuscule à la première lettre du nom du fichier, histoire d'être cohérent jusqu'au bout.

Vous devriez être capables de faire cela tous seuls avec votre IDE. Sous Code::Blocks, je passe par les menus File > New File , je saisirai par exemple le nom Personnage.h avec son extension et je réponds « Oui » quand Code::Blocks me demande si je veux ajouter le nouveau fichier au projet en cours (figure suivante).



### Personnage.h

Le fichier .h va donc contenir la déclaration de la classe avec les attributs et les prototypes des méthodes. Dans notre cas, pour la classe Personnage , nous obtenons :

```
1 #ifndef DEF_PERSONNAGE
2 #define DEF_PERSONNAGE
3
4 #include <string>
```

cpp

```

5
6 class Personnage
7 {
8     public:
9
10    void recevoirDegats(int nbDegats);
11    void attaquer(Personnage &cible);
12    void boirePotionDeVie(int quantitePotion);
13    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
14    bool estVivant();
15
16     private:
17
18     int m_vie;
19     int m_mana;
20     std::string m_nomArme; //Pas de using namespace std, il faut donc mettre std:: devant string
21     int m_degatsArme;
22 };
23
24 #endif

```

Comme vous pouvez le constater, seuls les prototypes des méthodes figurent dans le `.h`. C'est déjà beaucoup plus clair.

Dans les `.h`, il est recommandé de ne jamais mettre la directive `using namespace std;` car cela pourrait avoir des effets néfastes, par la suite, lorsque vous utiliserez la classe. Par conséquent, il faut rajouter le préfixe `std::` devant chaque `string` du `.h`. Sinon, le compilateur vous sortira une erreur du type `string does not name a type`.

### Personnage.cpp

C'est là qu'on va écrire le code de nos méthodes (on dit qu'on **implémente** les méthodes).

La première chose à ne pas oublier --sinon cela va mal se passer-- c'est d'inclure `<string>` et `Personnage.h`.

On peut aussi rajouter ici un `using namespace std;`. On a le droit de le faire car on est dans le `.cpp` (n'oubliez pas ce que je vous ai dit plus tôt : il faut éviter de le mettre dans le `.h`).

`cpp`

```

1 #include "Personnage.h"
2
3 using namespace std;

```

Maintenant, voilà comment cela se passe : pour chaque méthode, vous devez faire précéder le nom de la méthode par le nom de la classe suivi de deux fois deux points (:)). Pour `recevoirDegats`, voici ce que nous obtenons :

`cpp`

```

1 void Personnage::recevoirDegats(int nbDegats)
2 {

```

```

3
4 }

```

Cela permet au compilateur de savoir que cette méthode se rapporte à la classe Personnage . En effet, comme la méthode est ici écrite en dehors de la définition de la classe, le compilateur n'aurait pas su à quelle classe appartenait cette méthode.

## Personnage::recevoirDegats

Maintenant, c'est parti : implémentons la méthode recevoirDegats . Je vous avais expliqué un peu plus haut ce qu'il fallait faire. Vous allez voir, c'est très simple :

cpp

```

1 void Personnage::recevoirDegats(int nbDegats)
2 {
3     m_vie -= nbDegats;
4     //On enlève le nombre de dégâts reçus à la vie du personnage
5
6     if (m_vie < 0) //Pour éviter d'avoir une vie négative
7     {
8         m_vie = 0; //On met la vie à 0 (cela veut dire mort)
9     }
10 }

```

La méthode modifie donc la valeur de la vie. La méthode a le droit de modifier l'attribut , car elle fait partie de la classe. Ne soyez donc pas surpris : c'est justement l'endroit où on a le droit de toucher aux attributs.

La vie est diminuée du nombre de dégâts reçus. En théorie, on aurait pu se contenter de la première instruction mais on fait une vérification supplémentaire. Si la vie est descendue en-dessous de 0 (parce que le personnage a reçu 20 de dégâts alors qu'il ne lui restait plus que 10 de vie, par exemple), on ramène la vie à 0 pour éviter d'avoir une vie négative (cela ne fait pas très pro, une vie négative). De toute façon, à 0 de vie, le personnage est considéré comme mort.

Et voilà pour la première méthode ! Allez, on enchaîne !

## Personnage::attaquer

cpp

```

1 void Personnage::attaquer(Personnage &cible)
2 {
3     cible.recevoirDegats(m_degatsArme);
4     //On inflige à la cible les dégâts que cause notre arme
5 }

```

Cette méthode est peut-être très courante, elle n'en est pas moins très intéressante !

On reçoit en paramètre une référence vers un objet de type Personnage . On aurait pu recevoir aussi un pointeur mais, comme les références sont plus faciles à manipuler, on ne va pas s'en priver.

La référence concerne le personnage cible que l'on doit attaquer. Pour infliger des dégâts à la cible, on appelle sa méthode `recevoirDegats` en faisant : `cible.recevoirDegats`

Quelle quantité de dégâts envoyer à la cible ? Vous avez la réponse sous vos yeux : le nombre de points de dégâts indiqués par l'attribut `m_degatsArme` ! On envoie donc à la cible la valeur de `m_degatsArme` de notre personnage.

### Personnage::boirePotionDeVie

cpp

```
1 void Personnage::boirePotionDeVie(int quantitePotion)
2 {
3     m_vie += quantitePotion;
4
5     if (m_vie > 100) //Interdiction de dépasser 100 de vie
6     {
7         m_vie = 100;
8     }
9 }
```

Le personnage reprend autant de vie que ce que permet de récupérer la potion qu'il boit. On vérifie toutefois qu'il ne dépasse pas les 100 de vie car, comme on l'a dit plus tôt, il est interdit de dépasser cette valeur.

### Personnage::changerArme

cpp

```
1 void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)
2 {
3     m_nomArme = nomNouvelleArme;
4     m_degatsArme = degatsNouvelleArme;
5 }
```

Pour changer d'arme, on stocke dans nos attributs le nom de la nouvelle arme ainsi que ses nouveaux dégâts. Les instructions sont très simples : on fait simplement passer dans nos attributs ce qu'on a reçu en paramètres.

### Personnage::estVivant

cpp

```
1 bool Personnage::estVivant()
2 {
3     if (m_vie > 0) //Plus de 0 de vie ?
4     {
5         return true; //VRAI, il est vivant !
6     }
7     else
8     {
9         return false; //FAUX, il n'est plus vivant !
10    }
11 }
```

Cette méthode permet de vérifier que le personnage est toujours vivant. Elle renvoie vrai ( `true` ) s'il a plus de 0 de vie et faux ( `false` ) sinon.

On peut, cependant, faire plus court en renvoyant directement la valeur du test.

c\_cpp

```
1 bool Personnage::estVivant()
2 {
3     return m_vie > 0; //Renvoie true si m_vie > 0 et false sinon.
4 }
```

C'est plus concis et aussi beaucoup plus clair. Vous n'êtes peut-être pas encore habitué aux tests booléens, mais c'est la bonne manière d'écrire ce genre de fonctions. Les programmeurs ne veulent pas s'embêter avec des parenthèses inutiles.

## Code complet de `Personnage.cpp`

En résumé, voici le code complet de `Personnage.cpp` :

c\_cpp

```
1 #include "Personnage.h"
2
3 using namespace std;
4
5 void Personnage::recevoirDegats(int nbDegats)
6 {
7     m_vie -= nbDegats;
8     //On enlève le nombre de dégâts reçus à la vie du personnage
9
10    if (m_vie < 0) //Pour éviter d'avoir une vie négative
11    {
12        m_vie = 0; //On met la vie à 0 (cela veut dire mort)
13    }
14 }
15
16 void Personnage::attaquer(Personnage &cible)
17 {
18     cible.recevoirDegats(m_degatsArme);
19     //On inflige à la cible les dégâts que cause notre arme
20 }
21
22 void Personnage::boirePotionDeVie(int quantitePotion)
23 {
24     m_vie += quantitePotion;
25
26     if (m_vie > 100) //Interdiction de dépasser 100 de vie
27     {
28         m_vie = 100;
29     }
30 }
31
32 void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)
33 {
34     m_nomArme = nomNouvelleArme;
35     m_degatsArme = degatsNouvelleArme;
36 }
```

```
37  
38 bool Personnage::estVivant()  
39 {  
40     return m_vie > 0;  
41 }
```

### main.cpp

Retour au `main()`. Première chose à ne pas oublier : inclure `Personnage.h` pour pouvoir créer des objets de type `Personnage`.

cpp

```
1 #include "Personnage.h" //Ne pas oublier
```

Le `main()` reste le même que tout à l'heure, on n'a pas besoin de le modifier. Au final, le code est donc très court et le fichier `main.cpp` ne fait qu'utiliser les objets :

cpp

```
1 #include <iostream>  
2 #include "Personnage.h" //Ne pas oublier  
3  
4 using namespace std;  
5  
6 int main()  
7 {  
8     Personnage david, goliath;  
9     //Création de 2 objets de type Personnage : david et goliath  
10  
11     goliath.atttaquer(david); //goliath attaque david  
12     david.boirePotionDeVie(20); //david récupère 20 de vie en buvant une potion  
13     goliath.atttaquer(david); //goliath réattaque david  
14     david.atttaquer(goliath); //david contre-attaque... c'est assez clair non ?  
15     goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);  
16     goliath.atttaquer(david);  
17  
18     return 0;  
19 }
```

*N'exécutez pas le programme pour le moment.* En effet, nous n'avons toujours pas vu comment faire pour initialiser les attributs, ce qui rend notre programme inutilisable. Nous verrons comment le rendre pleinement fonctionnel au prochain chapitre et vous pourrez alors (enfin !) l'exécuter.

Pour le moment il faudra donc vous contenter de votre imagination. Essayez d'imaginer que David et Goliath sont bien en train de combattre (je ne veux pas vous gâcher la chute mais, normalement, c'est David qui gagne à la fin) !

## En résumé

- Il est nécessaire de créer une classe pour pouvoir ensuite créer des objets.

- La classe est le plan de construction de l'objet.
- Une classe est constituée d'attributs et de méthodes (variables et fonctions).
- Les éléments qui constituent la classe peuvent être publics ou privés. S'ils sont publics, tout le monde peut les utiliser n'importe où dans le code. S'ils sont privés, seule la classe peut les utiliser.
- En programmation orientée objet, on suit la règle d'encapsulation : on rend les attributs privés, afin d'obliger les autres développeurs à utiliser uniquement les méthodes.

DÉCOUVREZ LA NOTION DE  
◀ PROGRAMMATION ORIENTÉE OBJET  
(POO) ▶

CRÉEZ LES CLASSES (PARTIE 2/2)

## Les professeurs

### Mathieu Nebra

Entrepreneur à plein temps, auteur à plein temps et co-fondateur d'OpenClassrooms :o)

### Matthieu Schaller

Chercheur en astrophysique et cosmologie. Spécialiste en simulations numériques de galaxies sur superordinateurs.

## Découvrez aussi ce cours en...



Livre



PDF

OPENCLASSROOMS

ENTREPRISES

CONTACT

EN PLUS



Français

