

# Programmez avec le langage C++

50 heures  Difficile

Mis à jour le 06/02/2020



## Créez les classes (Partie 2/2)

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

Allez, on enchaîne ! Pas question de s'endormir, on est en plein dans la POO, là. Au chapitre précédent, nous avons appris à créer une classe basique, à rendre le code modulaire en POO et surtout nous avons découvert le principe d'encapsulation (je vous rappelle que l'encapsulation est très importante, c'est la base de la POO).

Dans ce chapitre, nous allons découvrir comment initialiser nos attributs à l'aide d'un **constructeur**, élément indispensable à toute classe qui se respecte. Puisqu'on parlera de constructeur, on parlera aussi de **destructeur**, vous verrez que cela va de pair.

Nous complèterons notre classe `Personnage` et nous l'associerons à une nouvelle classe `Arme` que nous allons créer. Nous découvrirons alors tout le pouvoir qu'offrent les combinaisons de classes et vous devriez normalement commencer à imaginer pas mal de possibilités à partir de là.

### Constructeur et destructeur



Reprenons. Nous avons maintenant 3 fichiers :

- `main.cpp` : il contient le `main()`, dans lequel nous avons créé deux objets de type `Personnage` : `david` et `goliath`.
- `Personnage.h` : c'est le header de la classe `Personnage`. Nous y faisons figurer les prototypes des méthodes et les attributs. Nous y définissons la portée (`public` / `private`) de chacun des éléments. Pour respecter le principe d'encapsulation, tous nos attributs sont privés, c'est-à-dire non accessibles de l'extérieur.

- **Personnage.cpp** : c'est le fichier dans lequel nous implémentons nos méthodes, c'est-à-dire dans lequel nous écrivons le code source des méthodes.

Pour l'instant, nous avons défini et implémenté pas mal de méthodes. Je voudrais vous parler ici de 2 méthodes particulières que l'on retrouve dans la plupart des classes : le constructeur et le destructeur.

- **le constructeur** : c'est une méthode appelée automatiquement à chaque fois que l'on crée un objet basé sur cette classe.
- **le destructeur** : c'est une méthode appelée automatiquement lorsqu'un objet est détruit, par exemple à la fin de la fonction dans laquelle il a été déclaré ou, si l'objet a été alloué dynamiquement avec `new`, lors d'un `delete`.

Voyons plus en détail comment fonctionnent ces méthodes un peu particulières...

## Le constructeur

Comme son nom l'indique, c'est une méthode qui sert à *construire* l'objet. Dès qu'on crée un objet, le constructeur est automatiquement appelé.

Par exemple, lorsqu'on écrit dans le `main()` :

cpp

```
1 Personnage david, goliath;
```

le constructeur de la classe `Personnage` est appelé pour créer l'objet `david` et une deuxième fois pour créer l'objet `goliath`.

Un constructeur par défaut est automatiquement créé par le compilateur. C'est un constructeur vide, qui ne fait rien de particulier.

On a cependant très souvent besoin de créer soi-même un constructeur qui remplace ce constructeur vide par défaut.

## Le rôle du constructeur

Si le constructeur est appelé lors de la création de l'objet, ce n'est pas pour faire joli. En fait, le rôle principal du constructeur est d'*initialiser* les attributs.

En effet, souvenez-vous : nos attributs sont déclarés dans `Personnage.h` mais ils ne sont pas initialisés !

Revoici le code du fichier `Personnage.h` :

cpp

```
1 #include <string>
2
3 class Personnage
4 {
```

```

5   public:
6
7     void recevoirDegats(int nbDegats);
8     void attaquer(Personnage &cible);
9     void boirePotionDeVie(int quantitePotion);
10    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
11    bool estVivant();
12
13
14  private:
15
16    int m_vie;
17    int m_mana;
18    std::string m_nomArme;
19    int m_degatsArme;
20 };

```

Nos attributs `m_vie`, `m_mana` et `m_degatsArmes` ne sont pas initialisés ! Pourquoi ? Parce qu'on n'a pas le droit d'initialiser les attributs ici. C'est justement dans le constructeur qu'il faut le faire.

En fait, le constructeur est indispensable pour initialiser les attributs qui ne sont pas des objets (ceux qui ont donc un type classique : `int`, `double`, `char` ...). En effet, ceux-ci ont une valeur inconnue en mémoire (cela peut être 0 comme -3451).

En revanche, les attributs qui sont des objets, comme c'est ici le cas de `m_nomArme` qui est un `string`, sont automatiquement initialisés par le langage C++ avec une valeur par défaut.

## Créer un constructeur

Le constructeur est une méthode mais une méthode un peu particulière.

En effet, pour créer un constructeur, il y a deux règles à respecter :

- Il faut que la méthode ait le même nom que la classe. Dans notre cas, la méthode devra donc s'appeler « Personnage ».
- La méthode ne doit rien renvoyer, pas même `void` ! C'est une méthode sans aucun type de retour.

Si on déclare son prototype dans `Personnage.h`, cela donne le code suivant :

cpp

```

1 #include <string>
2
3 class Personnage
4 {
5   public:
6
7     Personnage(); //Constructeur
8     void recevoirDegats(int nbDegats);
9     void attaquer(Personnage &cible);
10    void boirePotionDeVie(int quantitePotion);
11    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);

```

```

12     bool estVivant();
13
14
15     private:
16
17     int m_vie;
18     int m_mana;
19     std::string m_nomArme;
20     int m_degatsArme;
21 };

```

Le constructeur se voit du premier coup d'œil : déjà parce qu'il n'a aucun type de retour, ensuite parce qu'il porte le même nom que la classe.

Et si on en profitait pour coder ce constructeur dans `Personnage.cpp` ?

Voici à quoi pourrait ressembler son implémentation :

cpp

```

1 Personnage::Personnage()
2 {
3     m_vie = 100;
4     m_mana = 100;
5     m_nomArme = "Épée rouillée";
6     m_degatsArme = 10;
7 }

```

Notez que j'ai utilisé ici des accents. Ne vous en préoccupez pas pour le moment, j'y reviendrai.

Vous noterez une fois de plus qu'il n'y a pas de type de retour, pas même void (c'est une erreur que l'on fait souvent, c'est pourquoi j'insiste sur ce point).

J'ai choisi de mettre la vie et le mana à 100, le maximum, ce qui est logique. J'ai affecté par défaut une arme appelée « Épée rouillée » qui fait 10 de dégâts à chaque coup.

Et voilà ! Notre classe `Personnage` a un constructeur qui initialise les attributs, elle est désormais pleinement utilisable.

Maintenant, à chaque fois que l'on crée un objet de type `Personnage`, celui-ci est initialisé à 100 points de vie et de mana, avec l'arme « Épée rouillée ». Nos deux compères `david` et `goliath` commencent donc à égalité lorsqu'ils sont créés dans le `main()` :

cpp

```
1 Personnage david, goliath; //Les constructeurs de david et goliath sont appelés
```

## Autre façon d'initialiser avec un constructeur : la liste d'initialisation

Le C++ permet d'initialiser les attributs de la classe d'une autre manière (un peu déroutante) appelée **liste d'initialisation**. C'est une technique que je vous recommande d'utiliser quand vous le pouvez, c'est-à-dire presque toujours (c'est aussi la technique que nous utiliserons dans ce cours).

Reprenez le constructeur que nous venons de créer :

cpp

```

1 Personnage::Personnage()
2 {
3     m_vie = 100;
4     m_mana = 100;
5     m_nomArme = "Épée rouillée";
6     m_degatsArme = 10;
7 }
```

Le code que vous allez voir ci-dessous produit le même effet :

cpp

```

1 Personnage::Personnage() : m_vie(100), m_mana(100), m_nomArme("Épée rouillée"), m_degatsArme(10)
2 {
3     //Rien à mettre dans le corps du constructeur, tout a déjà été fait !
4 }
```

La nouveauté, c'est qu'on rajoute un symbole deux-points (😊) suivi de la liste des attributs que l'on veut initialiser avec, entre parenthèses, la valeur. Avec ce code, on initialise la vie à 100, le mana à 100, l'attribut `m_nomArme` à « Épée rouillée », etc.

Cette technique est un peu surprenante, surtout que, du coup, on n'a plus rien à mettre dans le corps du constructeur entre les accolades : tout a déjà été fait avant ! Elle a toutefois l'avantage d'être « plus propre » et se révélera pratique dans la suite du chapitre.

On utilisera donc autant que possible les listes d'initialisation avec les constructeurs, c'est une bonne habitude à prendre.

Le prototype du constructeur (dans le `.h`) ne change pas. Toute la partie qui suit les deux-points n'apparaît pas dans le prototype.

## Surchagez le constructeur

Vous savez qu'en C++, on a le droit de surcharger les fonctions, donc de surcharger les méthodes. Et comme le constructeur est une méthode, on a le droit de le surcharger lui aussi.

Pourquoi je vous en parle ? Ce n'est pas par hasard : en fait, le constructeur est une méthode que l'on a tendance à beaucoup surcharger. Cela permet de créer un objet de plusieurs façons différentes.

Pour l'instant, on a créé un constructeur sans paramètre :

cpp

```
1 Personnage();
```

On appelle cela : **le constructeur par défaut** (il fallait bien lui donner un nom, le pauvre).

Supposons que l'on souhaite créer un personnage qui ait dès le départ une meilleure arme... comment faire ?

C'est là que la surcharge devient utile. On va créer un deuxième constructeur qui prendra en paramètre le nom de l'arme et ses dégâts.

Dans `Personnage.h`, on rajoute donc ce prototype :

cpp

```
1 Personnage(std::string nomArme, int degatsArme);
```

Le préfixe `std::` est ici obligatoire, comme je vous l'ai dit plus tôt, car on n'utilise pas la directive `using namespace std;` dans le `.h` (je vous renvoie au chapitre précédent si vous avez un trou de mémoire).

L'implémentation dans `Personnage.cpp` sera la suivante :

cpp

```
1 Personnage::Personnage(string nomArme, int degatsArme) : m_vie(100), m_mana(100), m_nomArme(nomArme),  
m_degatsArme(degatsArme)  
2 {  
3  
4 }
```

Vous noterez ici tout l'intérêt de préfixer les attributs par « `m_` » : ainsi, on peut faire la différence dans le code entre `m_nomArme`, qui est un attribut, et `nomArme`, qui est le paramètre envoyé au constructeur.

Ici, on place simplement dans l'attribut de l'objet le nom de l'arme envoyé en paramètre. On recopie juste la valeur. C'est tout bête mais il faut le faire, sinon l'objet ne se « souviendra pas » du nom de l'arme qu'il possède.

La vie et le mana, eux, sont toujours fixés à 100 (il faut bien les initialiser) ; mais l'arme, quant à elle, peut maintenant être renseignée par l'utilisateur lorsqu'il crée l'objet.

### Quel utilisateur ?

Souvenez-vous : l'utilisateur, c'est celui qui crée et utilise les objets. Le concepteur, c'est celui qui crée les classes.

Dans notre cas, la création des objets est faite dans le `main()`. Pour le moment, la création de nos objets ressemble à cela :

cpp

```
1 Personnage david, goliath;
```

Comme on n'a spécifié aucun paramètre, c'est le constructeur par défaut (celui sans paramètre) qui sera appelé.

Maintenant, supposons que l'on veuille donner dès le départ une meilleure arme à Goliath ; on indique entre parenthèses le nom et la puissance de cette arme :

cpp

```
1 Personnage david, goliath("Épée aiguisée", 20);
```

Goliath est équipé dès sa création de l'épée aiguisée. David est équipé de l'arme par défaut, l'épée rouillée.

Comme on n'a spécifié aucun paramètre lors de la création de `david`, c'est le constructeur par défaut qui sera appelé pour lui. Pour `goliath`, comme on a spécifié des paramètres, c'est le constructeur qui prend en paramètre un `string` et un `int` qui sera appelé.

**Exercice :** on aurait aussi pu permettre à l'utilisateur de modifier la vie et le mana de départ mais je ne l'ai pas fait ici. Ce n'est pas compliqué, vous pouvez l'écrire pour vous entraîner. Cela vous fera un troisième constructeur surchargé.

## Le constructeur de copie

Je vous ai dit au début de ce chapitre, que le compilateur créait automatiquement un constructeur par défaut qui ne fait rien. Ce n'est pas tout, il crée aussi ce qu'on appelle un "constructeur de copie". C'est une surcharge du constructeur qui initialise notre objet en copiant les valeurs des attributs de l'autre objet.

Par exemple si l'on souhaite que `david` soit une copie conforme de `goliath`, il nous suffit d'écrire:

cpp

```
1 Personnage goliath("Épée aiguisée", 20); //On crée goliath en utilisant un constructeur normal
2
3 Personnage david(goliath); //On crée david en copiant tous les attributs de goliath
```

Ce constructeur est donc très simple à utiliser. Et comme je vous l'ai dit, le compilateur le crée automatiquement pour vous! C'est donc toute une partie du travail qui nous est épargnée. Merci le compilateur.

Si toute fois, vous désirez changer le comportement du constructeur de copie, il faut simplement le déclarer dans votre classe de la manière suivante:

cpp

```
1 Personnage(Personnage const& autre);
```

et de définir son implémentation comme suit:

cpp

```
1 Personnage::Personnage(Personnage const& autre): m_vie(autre.m_vie), m_mana(autre.m_mana),
m_nomArme(autre.m_nomArme), m_degatsArme(autre.m_degatsArme)
2 {
3 }
```

Vous remarquerez qu'on accède directement aux attributs de l'objet à copier (que j'ai appelé `autre`) dans la liste d'initialisation. C'est simple et concis.

## Le destructeur

Le destructeur est une méthode appelée lorsque l'objet est supprimé de la mémoire. Son principal rôle est de désallouer la mémoire (*via* des `delete`) qui a été allouée dynamiquement.

Dans le cas de notre classe `Personnage`, on n'a fait aucune allocation dynamique (il n'y a aucun `new`). Le destructeur est donc inutile. Cependant, vous en aurez certainement besoin un jour ou l'autre car on est souvent amené à faire des allocations dynamiques.

Tenez, l'objet `string` par exemple, vous croyez qu'il fonctionne comment ? Il a un destructeur qui lui permet, juste avant la destruction de l'objet, de supprimer le tableau de `char` qu'il a alloué dynamiquement en mémoire. Il fait donc un `delete` sur le tableau de `char`, ce qui permet de garder une mémoire propre et d'éviter les fameuses « fuites de mémoire ».

## Créez un destructeur

Bien que ce soit inutile dans notre cas (je n'ai pas utilisé d'allocation dynamique pour ne pas trop compliquer les choses tout de suite), je vais vous montrer comment on crée un destructeur. Voici les règles à suivre :

- Un destructeur est une méthode qui commence par un tilde (~) suivi du nom de la classe.
- Un destructeur ne renvoie aucune valeur, pas même `void` (comme le constructeur).
- Et, nouveauté : le destructeur ne peut prendre aucun paramètre. Il y a donc toujours un seul destructeur, il ne peut pas être surchargé.

Dans `Personnage.h`, le prototype du destructeur sera donc :

cpp

```
1 ~Personnage();
```

Dans `Personnage.cpp`, l'implémentation sera :

cpp

```
1 Personnage::~Personnage()
2 {
3     /* Rien à mettre ici car on ne fait pas d'allocation dynamique
4     dans la classe Personnage. Le destructeur est donc inutile mais
5     je le mets pour montrer à quoi cela ressemble.
6     En temps normal, un destructeur fait souvent des delete et quelques
7     autres vérifications si nécessaire avant la destruction de l'objet. */
8 }
```

Bon, vous l'aurez compris, mon destructeur ne fait rien. Ce n'était même pas la peine de le créer (il n'est pas obligatoire après tout).

Cela vous montre néanmoins la procédure à suivre. Soyez rassurés, nous ferons des allocations dynamiques plus tôt que vous ne le pensez et nous aurons alors grand besoin du destructeur pour désallouer la mémoire !

## Les méthodes constantes



Les méthodes constantes sont des méthodes de « lecture seule ». Elles possèdent le mot-clé `const` à la fin de leur prototype et de leur déclaration.

Quand vous dites « ma méthode est constante », vous indiquez au compilateur que votre méthode ne modifie pas l'objet, c'est-à-dire qu'elle ne modifie la valeur d'aucun de ses attributs. Par exemple, une méthode qui se contente d'afficher à l'écran des informations sur l'objet est une méthode constante : elle ne fait que lire les attributs. En revanche, une méthode qui met à jour le niveau de vie d'un personnage ne peut pas être constante.

On l'utilise ainsi :

cpp

```

1 //Prototype de la méthode (dans le .h) :
2 void maMethode(int parametre) const;
3
4
5 //Déclaration de la méthode (dans le .cpp) :
6 void MaClasse::maMethode(int parametre) const
7 {
8
9 }
```

On utilisera le mot-clé `const` sur des méthodes qui se contentent de renvoyer des informations sans modifier l'objet. C'est le cas par exemple de la méthode `estVivant()`, qui indique si le personnage est toujours vivant ou non. Elle ne modifie pas l'objet, elle se contente de vérifier le niveau de vie.

cpp

```

1 bool Personnage::estVivant() const
2 {
3     return m_vie > 0;
4 }
```

En revanche, une méthode comme `recevoirDegats()` ne peut pas être déclarée constante !

En effet, elle modifie le niveau de vie du personnage puisque celui-ci reçoit des dégâts.

On pourrait trouver d'autres exemples de méthodes concernées. Pensez par exemple à la méthode `size()` de la classe `string` : elle ne modifie pas l'objet, elle ne fait que nous informer de la longueur du texte contenu dans la chaîne.

Concrètement, à quoi cela sert-il de créer des méthodes constantes ?

Cela sert principalement à 3 choses :

- **Pour vous** : vous savez que votre méthode ne fait que lire les attributs et vous vous interdisez dès le début de les modifier. Si par erreur vous tentez d'en modifier un, le compilateur plante

en vous reprochant de ne pas respecter la règle que vous vous êtes fixée. Et cela, c'est bien.

- **Pour les utilisateurs de votre classe** : c'est très important aussi pour eux, cela leur indique que la méthode se contente de renvoyer un résultat et qu'elle ne modifie pas l'objet. Dans une documentation, le mot-clé `const` apparaît dans le prototype de la méthode et c'est un excellent indicateur de ce qu'elle fait, ou plutôt de ce qu'elle ne peut pas faire (cela pourrait se traduire par : « cette méthode ne modifiera pas votre objet »).
- **Pour le compilateur** : si vous rappelez le chapitre sur les variables, je vous conseillais de toujours déclarer `const` ce qui peut l'être. Nous sommes ici dans le même cas. On offre des garanties aux utilisateurs de la classe et on aide le compilateur à générer du code binaire de meilleure qualité.

## Associez des classes entre elles



La programmation orientée objet devient vraiment intéressante et puissante lorsqu'on se met à combiner plusieurs objets entre eux. Pour l'instant, nous n'avons créé qu'une seule classe :

`Personnage` .

Or en pratique, un programme objet est un programme constitué d'une multitude d'objets différents !

Il n'y a pas de secret, c'est en pratiquant que l'on apprend petit à petit à penser objet.

Ce que nous allons voir par la suite ne sera pas nouveau : vous allez réutiliser tout ce que vous savez déjà sur la création de classes, de manière à améliorer notre petit RPG et à vous entraîner à manipuler encore plus d'objets.

### La classe `Arme`

Je vous propose dans un premier temps de créer une nouvelle classe `Arme` . Plutôt que de mettre les informations de l'arme (`m_nomArme` , `m_degatsArme` ) directement dans `Personnage` , nous allons l'équiper d'un objet de type `Arme` . Le découpage de notre programme sera alors un peu plus dans la logique d'un programme orienté objet.

Souvenez-vous de ce que je vous ai dit au début : il y a 100 façons différentes de concevoir un même programme en POO. Tout est dans l'organisation des classes entre elles, la manière dont elles communiquent, etc.

Ce que nous avons fait jusqu'ici n'était pas mal mais je veux vous montrer qu'on peut faire autrement, un peu plus dans l'esprit objet, donc... mieux.

Qui dit nouvelle classe dit deux nouveaux fichiers :

- `Arme.h` : contient la définition de la classe ;
- `Arme.cpp` : contient l'implémentation des méthodes de la classe.

On n'est pas obligé de procéder ainsi. On pourrait tout mettre dans un seul fichier. On pourrait même mettre plusieurs classes par fichier, rien ne l'interdit en C++. Cependant, pour des raisons d'organisation, je vous recommande de faire comme moi.

## Arme.h

Voici ce que je propose de mettre dans Arme.h :

cpp

```
1 #ifndef DEF_ARME
2 #define DEF_ARME
3
4 #include <iostream>
5 #include <string>
6
7 class Arme
8 {
9     public:
10
11     Arme();
12     Arme(std::string nom, int degats);
13     void changer(std::string nom, int degats);
14     void afficher() const;
15
16     private:
17
18     std::string m_nom;
19     int m_degats;
20 };
21
22 #endif
```

Mis à part les `include` qu'il ne faut pas oublier, le reste de la classe est très simple.

On met le nom de l'arme et ses dégâts dans des attributs et, comme ce sont des attributs, on vérifie qu'ils sont bien privés (pensez à l'encapsulation). Vous remarquerez qu'au lieu de `m_nomArme` et `m_degatsArme`, j'ai choisi de nommer mes attributs `m_nom` et `m_degats` tout simplement. Si l'on y réfléchit, c'est en effet plus logique : on est déjà dans la classe `Arme`, ce n'est pas la peine de repréciser dans les attributs qu'il s'agit de l'arme, on le sait !

Ensuite, on ajoute un ou deux constructeurs, une méthode pour changer d'arme à tout moment, et une autre (allez, soyons fous) pour afficher le contenu de l'arme.

Reste à implémenter toutes ces méthodes dans `Arme.cpp`. Mais c'est facile, vous savez déjà le faire.

## Arme.cpp

Entraînez-vous à écrire `Arme.cpp`, c'est tout bête, les méthodes font au maximum deux lignes. Bref, c'est à la portée de tout le monde.

Voici mon Arme.cpp pour comparer :

cpp

```

1 #include "Arme.h"
2
3 using namespace std;
4
5 Arme::Arme() : m_nom("Épée rouillée"), m_degats(10)
6 {
7
8 }
9
10 Arme::Arme(string nom, int degats) : m_nom(nom), m_degats(degats)
11 {
12
13 }
14
15 void Arme::changer(string nom, int degats)
16 {
17     m_nom = nom;
18     m_degats = degats;
19 }
20
21 void Arme::afficher() const
22 {
23     cout << "Arme : " << m_nom << " (Dégâts : " << m_degats << ")" << endl;
24 }
```

N'oubliez pas d'inclure Arme.h si vous voulez que cela fonctionne.

## Et ensuite ?

Notre classe Arme est créée, de ce côté tout est bon. Mais maintenant, il faut adapter la classe Personnage pour qu'elle utilise non pas m\_nomArme et m\_degatsArme, mais un objet de type Arme. Et là... les choses se compliquent.

## Adaptez la classe Personnage pour utiliser la classe Arme

La classe Personnage va subir quelques modifications pour utiliser la classe Arme. Restez attentifs car utiliser un objet dans un objet, c'est un peu particulier.

### Personnage.h

Zou, direction le .h. On commence par enlever les deux attributs m\_nomArme et m\_degatsArme qui ne servent plus à rien.

Les méthodes n'ont pas besoin d'être changées. En fait, il vaut mieux ne pas y toucher. Pourquoi ? Parce que les méthodes peuvent déjà être utilisées par quelqu'un (par exemple dans notre main()). Si on les renomme ou si on les supprime, le programme ne fonctionnera plus.

Ce n'est peut-être pas grave pour un si petit programme mais, dans le cas d'un gros programme, si on supprime une méthode, c'est la catastrophe assurée dans le reste du programme. Et je ne vous

parle même pas de ceux qui écrivent des bibliothèques C++ : si, d'une version à l'autre des méthodes disparaissent, tous les programmes qui utilisent la bibliothèque ne fonctionneront plus !

Je vais peut-être vous surprendre en vous disant cela mais c'est là tout l'intérêt de la programmation orientée objet, et plus particulièrement de l'**encapsulation** : on peut changer les attributs comme on veut, vu qu'ils ne sont pas accessibles de l'extérieur ; on ne court pas le risque que quelqu'un les utilise déjà dans le programme.

En revanche, pour les méthodes, faites plus attention. Vous pouvez ajouter de nouvelles méthodes, modifier l'implémentation de celles existantes, mais pas en supprimer ou en renommer, sinon l'utilisateur risque d'avoir des problèmes.

Cette petite réflexion sur l'encapsulation étant faite (vous en comprendrez tout le sens avec la pratique), il faut ajouter un objet de type `Arme` à notre classe `Personnage`.

Il faut penser à ajouter un `include` de `Arme.h` si on veut pouvoir utiliser un objet de type `Arme`.

Voici mon nouveau `Personnage.h` :

cpp

```
1 #ifndef DEF_PERSONNAGE
2 #define DEF_PERSONNAGE
3
4 #include <iostream>
5 #include <string>
6 #include "Arme.h" //Ne PAS oublier d'inclure Arme.h pour en avoir la définition
7
8 class Personnage
9 {
10     public:
11
12     Personnage();
13     Personnage(std::string nomArme, int degatsArme);
14     ~Personnage();
15     void recevoirDegats(int nbDegats);
16     void attaquer(Personnage &cible);
17     void boirePotionDeVie(int quantitePotion);
18     void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
19     bool estVivant() const;
20
21
22     private:
23
24     int m_vie;
25     int m_mana;
26     Arme m_arame; //Notre Personnage possède une Arme
27 };
28
29 #endif
```

## Personnage.cpp

Nous n'avons besoin de changer que les méthodes qui utilisent l'arme, pour les adapter.

On commence par les constructeurs :

cpp

```

1 Personnage::Personnage() : m_vie(100), m_mana(100)
2 {
3
4 }
5
6 Personnage::Personnage(string nomArme, int degatsArme) : m_vie(100), m_mana(100), m_arame(nomArme,
degatsArme)
7 {
8
9 }
```

Notre objet `m_arame` est ici initialisé avec les valeurs reçues en paramètre par

`Personnage(nomArme, degatsArme)`. C'est là que la liste d'initialisation devient utile. En effet, on n'aurait pas pu initialiser `m_arame` sans une liste d'initialisation !

Peut-être ne voyez-vous pas bien pourquoi. Si je peux vous donner un conseil, c'est de ne pas vous prendre la tête à essayer de comprendre ici le pourquoi du comment. Contentez-vous de *toujours utiliser les listes d'initialisation avec vos constructeurs*, cela vous évitera bien des problèmes.

Revenons au code.

Dans le premier constructeur, c'est le constructeur par défaut de la classe `Arme` qui est appelé tandis que, dans le second, on appelle celui qui prend en paramètre un `string` et un `int`.

La méthode `recevoirDegats` n'a pas besoin de changer.

En revanche, la méthode `attaquer` est délicate. En effet, on ne peut pas faire :

cpp

```

1 void Personnage::attaquer(Personnage &cible)
2 {
3     cible.recevoirDegats(m_arame.m_degats);
4 }
```

Pourquoi est-ce interdit ? Parce que `m_degats` est un attribut et que, comme tout attribut qui se respecte, il est *privé* ! Diantre... Nous sommes en train d'utiliser la classe `Arme` au sein de la classe `Personnage` et, comme nous sommes utilisateurs, nous ne pouvons pas accéder aux éléments privés.

Comment résoudre le problème ? Il n'y a pas 36 solutions. Cela va peut-être vous surprendre mais on doit créer une méthode pour récupérer la valeur de cet attribut. Cette méthode est appelée **accesseur** et commence généralement par le préfixe « `get` » (« récupérer », en anglais). Dans notre cas, notre méthode s'appellerait `getDegats`.

On conseille généralement de rajouter le mot-clé `const` aux accesseurs pour en faire des méthodes constantes, puisqu'elles ne modifient pas l'objet.

cpp

```
1 int Arme::getDegats() const
2 {
3     return m_degats;
4 }
```

N'oubliez pas de mettre à jour `Arme.h` avec le prototype, qui sera le suivant :

cpp

```
1 int getDegats() const;
```

Voilà, cela peut paraître idiot et pourtant, c'est une sécurité nécessaire. On est parfois obligé de créer une méthode qui fait seulement un `return` pour accéder indirectement à un attribut.

De même, on crée parfois des accesseurs permettant de modifier des attributs. Ces accesseurs sont généralement précédés du préfixe « set » (« mettre », en anglais).

Vous avez peut-être l'impression qu'on viole la règle d'encapsulation ? Eh bien non. La méthode permet de faire des tests pour vérifier qu'on ne met pas n'importe quoi dans l'attribut, donc cela reste une façon sécurisée de modifier un attribut.

Vous pouvez maintenant retourner dans `Personnage.cpp` et écrire :

cpp

```
1 void Personnage::attaquer(Personnage &cible)
2 {
3     cible.recevoirDegats(m_arme.getDegats());
4 }
```

`getDegats` renvoie le nombre de dégâts, qu'on envoie à la méthode `recevoirDegats` de la cible.

Pfiou !

Le reste des méthodes n'a pas besoin de changer, à part `changerArme` de la classe `Personnage` :

cpp

```
1 void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)
2 {
3     m_arme.changer(nomNouvelleArme, degatsNouvelleArme);
4 }
```

On appelle la méthode `changer` de `m_arme`.

Le `Personnage` répercute donc la demande de changement d'arme à la méthode `changer` de son objet `m_arme`.

Comme vous pouvez le voir, on peut faire communiquer des objets entre eux, à condition d'être bien organisé et de se demander à chaque instant « est-ce que j'ai le droit d'accéder à cet élément

ou pas ? ».

N'hésitez pas à créer des accesseurs si besoin est : même si cela peut paraître lourd, c'est la bonne méthode. En aucun cas vous ne devez mettre un attribut `public` pour simplifier un problème. Vous perdriez tous les avantages et la sécurité de la POO (et vous n'auriez aucun intérêt à continuer le C++ dans ce cas).

## Action !



Nos personnages combattent dans le `main()`, mais... nous ne voyons rien de ce qui se passe. Il serait bien d'afficher l'état de chacun des personnages pour savoir où ils en sont.

Je vous propose de créer une méthode `afficherEtat` dans `Personnage`. Cette méthode sera chargée de faire des `cout` pour afficher dans la console la vie, le mana et l'arme du personnage.

## Prototype et include

On rajoute le prototype, tout bête, dans le `.h` :

cpp

```
1 void afficherEtat() const;
```

## Implémentation

Implémentons ensuite la méthode. C'est simple, on a simplement à faire des `cout`. Grâce aux attributs, on peut faire apparaître toutes les informations relatives au personnage :

cpp

```
1 void Personnage::afficherEtat() const
2 {
3     cout << "Vie : " << m_vie << endl;
4     cout << "Mana : " << m_mana << endl;
5     m_arme.afficher();
6 }
```

Comme vous pouvez le voir, les informations sur l'arme sont demandées à l'objet `m_arme` via sa méthode `afficher()`. Encore une fois, les objets communiquent entre eux pour récupérer les informations dont ils ont besoin.

## Appel de `afficherEtat` dans le `main()`

Bien, tout cela c'est bien beau mais, tant qu'on n'appelle pas la méthode, elle ne sert à rien. Je vous propose donc de compléter le `main()` et de rajouter à la fin les appels de méthode :

cpp

```
1 int main()
2 {
3     //Création des personnages
4     Personnage david, goliath("Épée aiguisée", 20);
5
6     //Au combat !
```

```
7     goliath.atttaquer(david);
8     david.boirePotionDeVie(20);
9     goliath.atttaquer(david);
10    david.atttaquer(goliath);
11    goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);
12    goliath.atttaquer(david);
13
14    //Temps mort ! Voyons voir la vie de chacun...
15    cout << "David" << endl;
16    david.afficherEtat();
17    cout << endl << "Goliath" << endl;
18    goliath.afficherEtat();
19
20    return 0;
21 }
```

On peut enfin exécuter le programme et voir quelque chose dans la console !

```
David
Vie : 40
Mana : 100
Arme : Épée rouillée (Degats : 10)
```

```
Goliath
Vie : 90
Mana : 100
Arme : Double hache tranchante vénéneuse de la mort (Degats : 40)
```

Si vous êtes sous Windows, vous aurez probablement un bug avec les accents dans la console. Ne vous en préoccuez pas, ce qui nous intéresse ici c'est le fonctionnement de la POO. Et puis de toute manière, dans la prochaine partie du livre, nous travaillerons avec de vraies fenêtres donc, la console, c'est temporaire pour nous. ;-)

Pour que vous puissiez vous faire une bonne idée du projet dans son ensemble, je vous propose de télécharger un fichier [zip](#) contenant :

- [main.cpp](#)
- [Personnage.cpp](#)
- [Personnage.h](#)
- [Arme.cpp](#)
- [Arme.h](#)

[Télécharger le projet RPG \(3 Ko\)](#)

Je vous invite à faire des tests pour vous entraîner. Par exemple :

- - Continuez à faire combattre `david` et `goliath` dans le `main()` en affichant leur état de temps en temps.
  - Introduisez un troisième personnage dans l'arène pour rendre le combat plus intéressant.
  - Rajoutez un attribut `m_nom` pour stocker le nom du personnage dans l'objet. Pour le moment, nos personnages ne savent même pas comment ils s'appellent, c'est un peu bête.

Du coup, je pense qu'il faudrait modifier les constructeurs et obliger l'utilisateur à indiquer un nom pour le personnage lors de sa création... à moins que vous ne donnez un nom par défaut si rien n'est précisé ? À vous de choisir !

- Rajoutez des `cout` dans les autres méthodes de `Personnage` pour indiquer à chaque fois ce qui est en train de se passer (« machin boit une potion qui lui redonne 20 points de vie »).
- Rajoutez d'autres méthodes au gré de votre imagination... et pourquoi pas des attaques magiques qui utilisent du mana ?
- Enfin, pour l'instant, le combat est écrit dans le `main()` mais vous pourriez laisser le joueur choisir les attaques dans la console à l'aide de `cin`.

Prenez cet exercice très au sérieux, ceci est peut-être la base de votre futur MMORPG (« Un jeu de rôle massivement multi-joueurs », si vous préférez) révolutionnaire !

Précision utile : la phrase ci-dessus était une boutade : ce cours ne vous apprendra pas à créer un MMORPG, vu le travail phénoménal que cela représente. Mieux vaut commencer par se concentrer sur de plus petits projets réalistes, et notre RPG en est un. Ce qui est intéressant ici, c'est de voir comment est conçu un jeu orienté objet (comme c'est le cas de la plupart des jeux aujourd'hui). Si vous avez bien compris le principe, vous devriez commencer à voir des objets dans tous les jeux que vous connaissez ! Par exemple, un bâtiment dans Starcraft 2 est un objet qui a un niveau de vie, un nom, il peut produire des unités (*via* une méthode), etc.

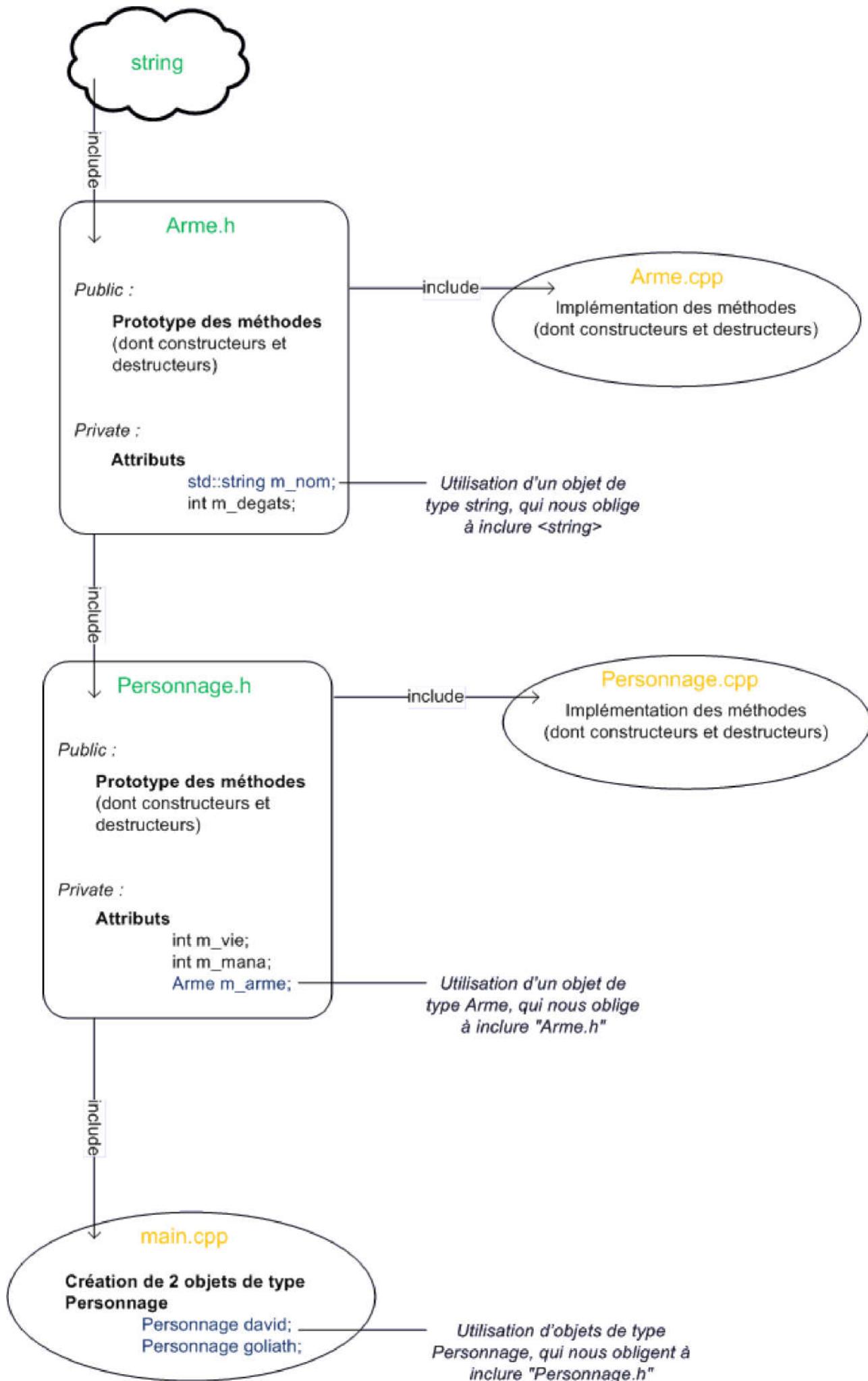
Si vous commencez à voir des objets partout, c'est bon signe ! C'est ce que l'on appelle « penser objet ».

## Méga schéma résumé



Croyez-moi si vous le voulez mais je ne vous demande même pas vraiment d'être capables de programmer tout ce qu'on vient de voir en C++. Je veux que vous reteniez le principe, le concept, comment tout cela est agencé.

Et pour retenir, rien de tel qu'un méga schéma bien mastoc, non ? Ouvrez grand vos yeux, je veux que vous soyez capables de reproduire la figure suivante les yeux fermés la tête en bas avec du poil à gratter dans le dos !



## Résumé de la structure du code

## En résumé

- Le constructeur est une méthode appelée automatiquement lorsqu'on crée l'objet. Le destructeur, lui, est appelé lorsque l'objet est supprimé.
- On peut surcharger un constructeur, c'est-à-dire créer plusieurs constructeurs. Cela permet de créer un objet de différentes manières.
- Une méthode constante est une méthode qui ne change pas l'objet. Cela signifie que les attributs ne sont pas modifiés par la méthode.
- Puisque le principe d'encapsulation impose de protéger les attributs, on crée des méthodes très simples appelées accesseurs qui renvoient la valeur d'un attribut. Par exemple, `getDegats()` renvoie la valeur de l'attribut `degats`.
- Un objet peut contenir un autre objet au sein de ses attributs.
- La programmation orientée objet impose un code très structuré. C'est ce qui rend le code souple, pérenne et réutilisable.

[CRÉEZ LES CLASSES \(PARTIE 1/2\)](#)[SURCHARGEZ UN OPÉRATEUR](#)

## 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](#)