POO = Programmation orienté objet

# Classes et objet (instance)

En POO, on va manipuler des objets.

Les objets sont en fait des instances d’une classe.

Exemple:

On peut par exemple en poo définir une Classe Personnage. En fait, cela correspond à un nouveau type de donnée.

Comme les entiers les flottant, les listes... maintenant on pourra utiliser notre type Personnage.

Dans un jeu la classe personnage sera instancié plusieurs fois.

Par exemple, notre héros sera un personnage, les ennemis seront des personnages, les autres héros seront des personnages, etc…

En fait, tous les éléments que nous avons vu, les personnages que l'on va créer les entiers que l'on a utilisés, les listes, ... sont des objets.

Souvent, un fichier ne contient qu’une seule classe, mais il arrive que l’on ait également des classes internes.

Exemple:

Déclaration d’une classe:

class Personnage

{

//la définition des propriétés et méthodes de notre personnage

}

Une fois que l'on a défini nos classes. On peut définir des instances de cette classe. C'est-à-dire des objets typés par cette classe.

Exemple:

[1,2] est un objet de type liste

monHeros est un objet (une instance) du type personnage

Instanciation (Déclaration d’une instance ou d’un objet) en utilisant le constructeur de la classe:

Personnage david, goliath();

## 1.1 class ou struct:

On peut également définir de nouveau type avec le mot clé struct à la place de class. La différence

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

# Attributs

Lors de la définition de notre classe personnage.

On va définir les attributs qu'un personnage devra avoir. Les attributs sont les propriétés que devront avoir les éléments de la classe que l’on est en train de définir.

Par exemple:

Force, magie, résistance, Vie, RM, Mana, …

class Personnage

{

private:

int vie;

int mana;

Arme arme;

static int niveauMax=99

}

# Méthode

Un personnage doit aussi pouvoir exécuter plusieurs actions. Ces actions vont être définies dans des méthodes (fonctions) spécifiques à nos personnages.

Les méthodes sont les fonctions que l’on veut pouvoir effectuer sur nos classes.

La déclaration des méthodes est différente entre le header (.h) et le code (.cpp)

Exemple:

Attaquer

Esquiver

Parler

Lancer sort

...

Dans le header:

class Personnage

{

public:

void recevoirDegats(int nbDegats);

void attaquer(Personnage &cible);

void boirePotionDeVie(int quantitePotion);

void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);

bool estVivant();

void afficherEtat();

private:

int vie;

int mana;

Arme arme;

};

Dans le cpp:

Personnage::Personnage() : vie(100), mana(100)

{

}

Personnage::Personnage(string nomArme, int degatsArme) : m\_vie(100), mana(100), arme(nomArme, degatsArme)

{

}

void Personnage::attaquer(Personnage &cible)

{

cible.recevoirDegats(arme.getDegats());

}

void Personnage::boirePotionDeVie(int quantitePotion)

{

vie += quantitePotion;

if (vie > 100)

{

vie = 100;

}

}

void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)

{

arme.changer(nomNouvelleArme, degatsNouvelleArme);

}

bool Personnage::estVivant()

{

if (vie > 0)

{

return true;

}

else

{

return false;

}

}

## 3.1 Constructeur

Les constructeurs sont les méthodes qui servent à créer des instances du type de la classe qu’on est en train de créer.

Lorsque l’on créer une classe, un constructeur est automatiquement initialisé pour pouvoir permettre de créer de nouveaux personnages. Seulement, ce constructeur par défaut ne sait pas quelle valeur il devra donner aux attributs de notre personnage (quelle force ?, quelle vie? et quel niveau?)

Ici Personnage est une méthode de ma classe Personnage qui sert à créer des nouveaux personnages. Pour une même classe on peut définir **plusieurs constructeurs avec des paramètres différents** à chaque fois.

Dans le header (.h)

class Personnage

{

public:

Personnage();

Personnage(int vieInit, int forceInit, int niveauInit);

Personnage(int niveau);

Ici, on déclare plusieurs constructeurs, certains s’utilisent avec des paramètres alors que d’autres non.

Dans le cpp, il existe deux manière différente de faire cela:

* En utilisant la liste d’initialisation:

Personnage::Personnage(int vieInit, int forceInit, int niveauInit) : vie(vieInit), force(forceInit), niveau(niveauInit)

{

}

Personnage::Personnage(int niveauInit) : vie(niveauInit\*100), force(niveauInit\*10), niveau(niveauInit)

{

}

Personnage::Personnage() : vie(100), force(10), niveau(1)

{

}

* Ou de manière un peu plus lisible (et plus courante):

Personnage::Personnage()

{

vie=100;

force=10;

niveau=1;

}

Personnage::Personnage(int vieInit, int forceInit, int niveauInit)

{

vie=vieInit;

force=forceInit;

niveau=niveauInit;

}

Pour utiliser un constructeur avec paramètres ou sans paramètre:

Personnage heros1(100, 10, 1);

Personnage heros2(1);

Personnage heros3();

Un constructeur sans paramètre existe par défaut dès qu’on définit une classe, mais généralement on préfère le définir, au moins pour spécifier sa portée.

### 3.1.2 Le constructeur de copie

En plus du constructeur par défaut que l’on a vu plus tôt, 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.

Personnage heros1(100, 10, 1);

Personnage heros2(heros1);

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

Dans h++:

Personnage(Personnage const& autre);

Dans cpp:

Personnage::Personnage(Personnage const& autre): vie(autre.vie), force(autre.force),niveau(autre.niveau)

{

}

## 3.2 Le destructeur

Pour détruire un personnage dont on ne va plus se servir et libérer de la place dans la mémoire, on utilise un destructeur.

* un destructeur porte le même nom que la classe dans laquelle il est défini et est précédé d'un tilde ~.
* un destructeur n'a **pas de type de retour** (même pas *void*)
* un destructeur **ne peut pas** avoir d'argument
* la définition d'un destructeur n'est **pas obligatoire** lorsque celui-ci n'est pas nécessaire

Il se déclare seulement dans le header (.h):

{

public:

~Personnage();

Forcer l’appel d’un destructeur:

s.String::~String(); // non-virtual call

ps->String::~String(); // non-virtual call

s.~String(); // Virtual call

ps->~String(); // Virtual call

Les destructeurs sont appelés lorsque l'un des événements suivants se produit :

* Un objet (automatique) local avec portée de bloc passe hors de portée.
* Un objet alloué à l’aide de l' new opérateur est désalloué explicitement à l’aide de delete .
* La durée de vie d'un objet temporaire se termine.
* Un programme se termine et des objets globaux ou statiques existent.
* Le destructeur est appelé explicitement à l'aide du nom complet de la fonction destructeur.

On pourrait redéfinir le destructeur pour qu’il détruise également l’arme associée au personnage.

## 3.3 Attribut et méthode static (par opposition au dynamique).

Les méthodes et les attributs static sont liées à la classe plutôt qu’à leurs instances.

Exemple sur notre classe personnage, le constructeur est une méthode static.

Le nombre de PV limite est un attribut static de la classe car il sera le même pour toutes les instances de personnages (attention, pv != pv max != pv limite).

* **Déclaration**:

Header:

private:

static int maxLevel;

std::string name;

public:

static int getMaxLevel() const;

std::string getName const;

Cpp

Personnage perso();

Personnage::getMaxLevel()

perso.getName();

Remarque: Une méthode est constante (const) lorsqu’elle ne modifie pas l’objet sur lequel elle est appelée.

Il est préférable de bien indiquer lorsqu’une méthode est constante pour déclarer à l’utilisateur qu’il ne risque pas de modifier son objet. Et cela permet également de préciser aux classes filles de la classe qui vont réimplémenter cette méthode qu’elle doit rester constante.

* **Utilisation**:

Une méthode static s’utilise de la façon suivante:

int maxlevel=Personnage::getMaxLevel();

//Les méthodes non static en revanche s'utilisent comme suit:

int herosLvl=heros.getLevel();

## 3.4. portée des méthodes et attributs.

### **3.4.1 public**, **private** et **protected**

Pour rendre accessible ou caché ses attributs et ces méthodes dans les autres classes. On doit définir une visibilité pour chacun des attributs et méthodes.

Le mot clé public sert à définir qu'un attribut/méthode sera visible par tous.

**private**

Vu de l'extérieur de la classe Inventaire. La méthode canAddObject ne peut pas être utilisée. En fait, cette méthode sert juste à simplifier la structure du code de nos autres méthodes de la classe inventaire.

**public**

Par contre, la méthode addObject sera utilisable sur n'importe quel objet de la classe inventaire et ce depuis n'importe quel autre classe.

**protected**

Protected est utilisé comme private mais pour les attributs et methode que l' on veut aussi que les classes filles (voire heritage) puissent utiliser.

### 3.4.2 Méthode ami: friend

*Une fonction amie d'une classe peut accéder à tous ses éléments, même les éléments privés.*

*Vos programmes devraient respecter les deux règles suivantes :*

* *une fonction amie ne doit pas, en principe, modifier l'instance de la classe ;*
* *les fonctions amies ne doivent être utilisées que si vous ne pouvez pas faire autrement.*

*Cette deuxième règle est très importante. Si vous ne la respectez pas, alors autant arrêter la POO car le concept de classe perd tout son sens.*

Exemple:

class B;

class A {

public:

int Func1( B& b );

int Func2( B& b );

};

class B {

private:

int \_b;

// A::Func1 is a friend function to class B

// so A::Func1 has access to all members of B

friend int A::Func1( B& );

};

int A::Func1( B& b ) { return b.\_b; } // OK

int A::Func2( B& b ) { return b.\_b; } // membre "B::\_b" est inaccessibleC/C++(265)

### **3.4.3 Résoudre les problèmes de dépendances à la compilation (le f**orward)

Dans certains cas, deux classes peuvent dépendre chacune l’une de l’autre. Par exemple, notre classe Personnage doit inclure la classe Arme, mais la classe Arme aurait très bien pu dépendre elle aussi du personnage. Dans ce cas, on ne peut pas utiliser un include pour chaque classe. Sinon on introduit une dépendance cyclique.

Pour résoudre ce problème, on va préciser avant de déclarer la classe Arme que la classe Personnage existe.

class Personnage ;

class Arme {

public:

...

};

### Encapsulation

L’encapsulation est un principe de la POO, qui permet de protéger les propriétés d’une classe.

Pour cela, on met l’attribut de la classe en private.

### 4.1 Accesseurs et mutateurs (getters et setters)

Pour protéger les propriétés d’une classe, on les met généralement en private. Par contre on définit des méthodes permettant de modifier (setter) la valeur de cet attribut ou d’accéder (de lire) cette valeur.

Header:

class Personnage{

private :

int level;

public :

int GetLevel();

};

Cpp:

int Personnage::GetLevel(){

return level;

}

# Héritage

Maintenant, si on a besoin d’étendre la classe personnage en ajoutant de nouvelles méthodes ou de nouveaux attributs.

On peut créer une nouvelle classe (Guerrier, Mage) qui hérite des attributs de personnage avec des méthodes ou des attributs en plus.

On peut par exemple ajouter de nouveaux attributs ou de nouvelles méthodes à notre nouvelle classe.

Dans le header (.h)

class Mage:Personnage

{

private:

int magie=0;

}

## 5.1 Constructeur

Par défaut, notre classe magicien va utiliser le constructeur défini par défaut dans sa classe mère.

Elle va donc créer un Mage avec 100 de vie, 10 forces et de niveau 1.

Par contre elle ne saura pas quelle magie il doit avoir à la construction.

Mage::Mage (){

}

Il faut donc définir un nouveau constructeur pour ça.

Mage::Mage() : Personnage(), magie(10)

{

}

## 5.2 Overriding / Redéfinition

Par défaut, les attributs et les méthodes que l’on avait défini dans notre classe Personnage seront toutes disponibles dans la classe Mage.

Certaines implémentations de méthodes de la classe personnage ne sont plus pertinentes pour un mage.

Par exemple, on peut penser qu’un Mage n’aura pas la même façon d’attaquer que les autres personnages, son attaque utilisera la magie plutôt que la force par exemple.

Dans ce cas, il faudra redéfinir cette méthode attaquer et coder une implémentation spécifique à notre classe Mage.

C’est ce qu’on appelle la redéfinition (overriding).

void attaquer(Personnage &cible);

void Mage::attaquer(Personnage &cible){

cible.recevoirDegats(magie);

}

On dit aussi que la méthode attaquer de personnage est masqué par la méthode attaquer du Mage.

## 5.3 Surcharge

Il est possible qu’une fonction ne doive pas avoir le même comportement selon le type d’élément que l’on met en paramètre.

Par exemple, on pourrait avoir une méthode attaquer qui n’ai pas le même comportement

void Mage::attaquer(Personnage &cible){

cible.recevoirDegats(magie);

}

Admettons qu’on veuille quelque chose du genre:

void Mage::attaquer(Mage &cible){

cible.recevoirDegats(magie-cible.magie);

}

Dans ce cas, selon le paramètre (la cible) que l’on va attaquer, on utilisera la deuxième si c’est un Mage et la première si c’est une autre sorte de personnage.

## 5.4 Méthodes virtuelles et classes abstraites

Une méthode virtuelle est une méthode qui doit **IMPÉRATIVEMENT** être implémentée dans les sous classes.

virtual void Personnage::attaquer(Personnage &cible);//virtual function.

Une méthode abstraite (virtuelle pure) est une méthode virtuelle dans lequel on ne définit pas d’implémentation par défaut.

On utilise la syntaxe = 0 pour cela.

virtual void Personnage::attaquer() = 0; // Pure virtual function.

Dans les deux cas, quand on va définir les classes filles Guerrier, Mage, Archer, il faudra que chacune de ces classes implémentent cette méthode attaquer.

Lorsqu’une classe possède au moins une méthode virtuelle pure. On ne pourra pas instancier cette classe.

On dit alors que cette classe est **abstraite**.

*Cette notion n’est pas explicitement présente en C++ contrairement à certain langage qui utilise le mot clé abstract, pour cela, on va juste définir les méthodes de cette classe comme étant virtuelles.*

Dans les classes filles, on aura forcément:

virtual void Mage::attaquer(Personnage &cible);

## 5.5 Héritage virtuel

*On parle d'héritage virtuel lorsque la classe de base est précédée du modificateur virtual comme ci-dessous. Ce type d'héritage est uniquement utile lors d'héritage multiple où une classe de base apparaît de manière multiple.*

// Classe de Base

class B {

...

};

// Classe dérivée avec héritage virtuel

class D : public virtual B {

...

};

class C : virtual public A

## 5.6 Héritage multiple (à utiliser avec précaution)

*L’héritage multiple est possible en C++ mais il faut l’utiliser avec précaution.*

class MageGuerrier: Mage, Guerrier

{

*En effet, il y a un risque de conflit entre les méthodes et attributs des deux classes. Lesquelles privilégier si les deux ne sont pas identiques?*

*Par exemple, si Mage et Guerrier possède une méthode attaquer et que ces deux méthodes sont différentes, on ne peut pas savoir laquelle des deux pourra être utilisée par notre MageGuerrier.*

# Association

## 6.1 Les pointeurs (et gestion de la mémoire):

Chaque variable que l’on crée est stockée à un endroit spécifique de la mémoire de votre ordinateur. La “case” dans laquelle est stockée une variable s’appelle son adresse.

Adresse:

Pour obtenir l'adresse d’une variable, on peut utiliser &.

Par exemple: &var va nous donner l’adresse de la variable var.

Un pointeur est en fait une nouvelle variable mais qui va contenir la valeur d’une adresse d’une autre variable.

Déclarez un pointeur

Pour déclarer un pointeur il faut, comme pour les variables, deux choses :

* un type ;
* un nom.

Pour le nom, il n'y a rien de particulier à signaler. Les mêmes règles que pour les variables s'appliquent.

int \*pointeur(0);

int\* pointeur(0);

Les deux notations sont équivalentes.

Attention:

Il faut toujours utiliser les pointeurs en précisant une valeur.

Si on fait seulement:

int\* pointeur;

Le pointeur va pointer vers une adresse de la mémoire aléatoirement, ce qui peut être dangereux.

Exemple:

int main()

{

int ageUtilisateur(19);

//Une variable de type int

int \*ptr(0);

//Un pointeur pouvant contenir l'adresse d'un nombre entier

ptr = &ageUtilisateur;

//On met l'adresse de 'ageUtilisateur' dans le pointeur 'ptr'

cout << "La valeur de la variable ageUtilisateur est: " << \*ptr << endl;

return 0;

}

Attention:

Pour bien utiliser les pointeurs il faut allouer de la mémoire et pensez à la libérer quand on ne s’en sert plus.

#include <iostream>

using namespace std;

int main()

{

int\* pointeur(0);

pointeur = new int;

cout << "Quel est votre age ? ";

cin >> \*pointeur;

//On écrit dans la case mémoire pointée par le pointeur 'pointeur'

cout << "Vous avez " << \*pointeur << " ans." << endl;

//On utilise à nouveau \*pointeur

delete pointeur; //Ne pas oublier de libérer la mémoire

pointeur = 0; //Et de faire pointer le pointeur vers rien

return 0;

}

## 6.2 Association:

Revenons à la POO.

Il existe plusieurs types de liens entre les différentes classes. On distingue généralement les agrégations et les compositions.

**Agrégation ou composition:**



Une entreprise utilise plusieurs camions mais un camion peut très bien exister sans entreprise. On a donc affaire à une **agrégation**.

Pour représenter une agrégation:

Arme \*m\_arme; //L'Arme est un pointeur, l'objet n'est plus contenu dans le Personnage

Par contre, un camion contient un moteur, c’est donc une **composition**. Si on détruit le camion, il faut aussi détruire son moteur.

Pour représenter une composition:

Arme arme; // L'Arme est "contenue" dans le Personnage

Lorsqu’on utilise un pointeur, on pointe vers une arme qui existe indépendamment de notre personnage. Dans le cas contraire, l’arme n’est accessible que pour ce personnage.

**Impact de passer d’une composition à une agrégation:**

**Il faut changer le constructeur:**

Notre Arme étant un pointeur, il va falloir le créer par le biais d'une allocation dynamique avec new. Sinon, l'objet ne se créera pas tout seul.

Pour cela, on va faire:

**Personnage::Personnage(int degatsArme) : arme(0),vie(100), force(100)**

**{**

**arme = new Arme(degatsArme);**

**}**

**Le destructeur:**

Notre Arme étant maintenant un pointeur, lorsque l'objet de type Personnage est supprimé, l'Arme ne disparaît pas toute seule ! Si on se contente d'un **new** dans le constructeur, et qu'on ne met rien dans le destructeur, lorsque l'objet de type Personnage est détruit, nous aurons une partie de la mémoire qui ne sera pas libérée pour rien.

**Personnage::~Personnage()**

**{**

**delete m\_arme;**

**}**

**Les méthodes:**

Enfin, il faut également mettre à jour les méthodes qui utilisent cette arme.

void Personnage::attaquer(Personnage &cible)

{

cible.recevoirDegats(m\_arme.getDegats());

}

… devient :

void Personnage::attaquer(Personnage &cible)

{

cible.recevoirDegats(m\_arme->getDegats());

}

**Multiplicité:**

La multiplicité associée à une terminaison d'association, d'agrégation ou de composition déclare le nombre d'objets susceptibles d'occuper la position définie par la terminaison d'association. Voici quelques exemples de multiplicité :

* exactement un : 1 ou 1..1 ;
* plusieurs : \* ou 0..\* ;
* au moins un : 1..\* ;
* de un à six : 1..6.

# 7. Autres mots clés

* **Mot clé final sur les attributs, classes et méthodes.**

Sur une classe, cela signifie que la classe ne pourra pas être étendue.

Exemple:

final class Mage{}

Sur une méthode, cela signifie que cette méthode ne pourra pas être étendue (à l’inverse du virtual). Le mot clé final signifie qu’une classe, un attribut ou une méthode ne peut pas être redéfini.

* **Attribut dérivé**

Exemple: degat\_arme sur un personnage peut être calculé en allant recupéré l’arme et la valeur de son attribut\_dégat.

Quand on change d'arme, il faudra notifier à l’utilisateur qu’il doit changer cette valeur.

Ce type d’attribut permet généralement de garder en mémoire le résultat d’un calcul afin de pouvoir y accéder plus rapidement.

* **Const ou static**

Un attribut constant est un attribut qui ne pourra jamais être changé.

Un attribut static est un attribut qui est propre à la classe dans laquelle il est défini (et non pas à ces instances).

Exemple un attribut niveauMax sur notre classe Personnage, on veut que l’attribut niveauMax soit porté par la classe de sorte que si on le change alors ce niveauMax s’appliquera pour tous nos personnages, pas seulement une instance.

Une méthode est constante lorsqu’elle ne modifie pas l’objet sur lequel elle est appelée.

Une méthode est statique si on veut pouvoir l’appeler directement à partir de la classe (et pas depuis une instance).

# **8. Conversion de type** (**Caster)**:

Pour vérifier qu’un objet est bien du type souhaité ou pour le convertir dans un type donné. On va devoir caster cet élément.

Pour cela, on va généralement utiliser un static\_cast, mais il existe d’autres types de cast.

**static\_cast:**

On utilise le static\_cast lorsqu’on connaît le type de l’objet que l’on veut caster et le type dans lequel on veut pouvoir le caster. (Exemple: un double peut toujours être caster en flottant, de même qu’un Mage peut toujours être caster en personnage).

double d;

float f = **static\_cast**<float>(d);

**reinterpret\_cast:**

Le réinterpreter\_cast quant à lui va permettre de caster des objets qui ne devrait pas pouvoir être caster. (Ici en castant un double en flottant, on peut s’exposer à des erreurs lors de l'exécution).

En effet, ici c’est comme si on voulait absolument caster un Personnage en Mage. Mais attention, tous les personnages ne sont pas des mages.

double\* fp dp;

float\* = **reinterpret\_cast**<float\*>(dp);

**const\_cast:**

S’utilise avec des constantes.

**dynamic\_cast:**

On utilise le dynamic cast lorsque l’on est pas sûr de devoir caster l’objet.

|  |
| --- |
| **class** **polygone**  {  **public** :  **virtual** void f() {}  };  **class** **carre** : **public** polygone {};  int main()  {  carre monCarre;  carre& r\_carre = monCarre;  try {  polygone& r\_polygone = **dynamic\_cast**<polygone&>(r\_carre);  }  **catch** (**const** std::exception& e)  {  std::cerr << e.what();  }  **return** EXIT\_SUCCESS;  } |

<https://zestedesavoir.com/tutoriels/553/les-conversions-de-types-en-c/>

* Vérifier le type d’un objet:

On peut également vouloir tester si notre objet est d’un type particulier.

Par exemple, dans une méthode attaquer(Personnage), il peut être intéressant de savoir si le Personnage que l’on attaque est un archer, un Mage ou un Guerrier.

Pour cela on utilise le

is\_base\_of<Base, T>

# **9. Compilation**:

Step 1 : Installer Mingw64

Step 2 : Ajouter le chemin de l’install dans la variable d’environnement PATH.

Step 3 : Tester l’installation

Step 4 : Builder

Lorsque l’on demande au compilateur de builder main.cpp. Il va parcourir ces dépendances pour voir s’il peut également compiler celles-ci.

Problème: Dans notre cas, main dépend de “Arme.h” et de “Perso.h”.

De plus, "Perso.h" dépend également de “Arme.h”.

En fait, il va essayer de builder deux fois Arme.h et va trouver des conflits entre les deux.

Pour éviter cela, on précise dans l'entête du fichier que ce fichier ne doit être compilé qu’une fois. Pour éviter cela, il y a deux possibilités:

#ifndef ARME\_H

#define ARME\_H

**class Arme** {

…

};

#endif */\* ARME\_H\*/*

Ou plus simplement:

#pragma once

Attention, la première solution est tout de plus efficace (performance), mais la deuxième reste plus simple à écrire.