Informatique

Travaux Pratiques

Axel LE BOT

Table des matières

Table des figures

Shell

1. TP1+TP2 : Shell

1.1 Manipulations de l'environnement et des fichiers sous UNIX

1.1.1 Exercice 1 : Découverte de quelques commandes d'archivage

L'objectif de cet exercice est de découvrir et manipuler les commandes de téléchargement, d'archivage, de compression et de décompression de fichier

1. Récupération et décompression d'une archive

La commande `wget <url>` permet de télécharger un fichier présent à cette adresse. Ici nous récupérons une archive que nous pouvons manipuler avec `tar`. `tar` a trois options intéressante :

- L'option `-x` permet de restaurer les fichiers contenus dans une archive.
- L'option `-c` permet de créer une nouvelle archive.
- L'option `-f` permet d'utilise le fichier archive F ou le périphérique F (par défaut /dev/rmt0).

On découvre que l'archive télécharger contient 9 fichiers.

2. Manipulation de fichiers

- La commande `file <filename>` nous permet de savoir le format d'un fichier.
- La commande `mv <filename> <filename2>` me permet de déplacer mais aussi de renommer un "fichier1" en "fichier2". Ici j'ai utilisé la commande suivante : `mv image4.jpg image4.jpg2 .`
- La commande `ls -lh <filename>` permet d'afficher les informations plus détaillé lisible par l'humain. Ici la commande nous apprend que le fichier `script.txt` fait 170Ko.
- La commande `gzip <filename>` permet de compresser un fichier. Ici le fichier `script.txt` a été compressé. Il fait maintenant 65Ko. La compréssion est donc d'environ 38.235%.

— La commande `gunzip <filename>` permet de décompresser un fichier. Ici le fichier `script.txt` fait maintenant 170Ko, qui est bien la taille initial du fichier.

3. Création d'une nouvelle archive

— La commande `tar` permet de créer une archive sans la compresser. On peut tout de même utiliser `tar` pour archiver puis compresser des fichiers
— La commande `tar -z` permet de compréssser l'archive au format `gzip`.
— La commande `tar -cz *.jpg *.txt *.jp2` n'est pas exécutée car il est impossible d'écrire des données compressées dans le terminal. Pour sela il faut rajouter l'option `-f`.
— La commande `tar -cz *.jpg *.txt *.jp2 > nouvelleArchive3.tar.gz` redirige bien le résultat dans un fichier. En effet le symbole > permet de rediriger la sortie vers un fichier. symbole >.

La redirection du flux dans un fichier recréer une archive compréssé "archive3" similaire à "archive2" créé. En conclusion l'archive 2 et 3 donne le même résultat et sont plus petit que l'archive 1 puisqu'elles sont compréssés.

1.1.2 Exercice 2 : Utilisation des masques de création de fichiers

Cet exercice permet de comprendre les droits UNIX à la création de fichier à l'aide des masque utilisateur. La commande `umask` permet de définir les permissions par défaut de fichiers ou de répertoires créees. Pour sela il faut préciser les droits que l'on veut supprimer. Ci-dessous un tableau de droits UNIX pour rappel.

Humain	Base 8	Base 2
---	0	000
--x	1	001
-w-	2	010
-wx	3	011
r--	4	100
r-x	5	101
rw-	6	110
rwx	7	111

FIGURE 1.1 – Permissions Unix

1.

Afin de créer des fichiers avec les droits adecquates, il faudra effectuer les commandes suivantes :

```
touch Raphael.txt
umask 0666
touch Donatello.txt
umask 0331
touch Michelangelo.txt
```

```
umask 0661
touch Leonardo.txt
umask 0000
```

2. et 3.

Après plusieurs test, on découvre qu'il n'est pas possible de donner plus de droit que la limitation par défaut du systeme.

— `umask 666` sur les fichiers
— `umask 777` sur les répertoires

1.1.3 Exercice 3 : Manipulation du Systeme de fichier et des droits de navigation

L'objectif de cet exercice est de manipuler les commandes de navigation dans l'arborescence et de création de fichier.

— La commande `mkdir <dirname>` permet de créer un répertoire.

1.

L'archive contient 5 images.

3.

Le chemin absolue d'un fichier ou d'un repertoire correspond au chemin vers le fichier ou repertoire depuis la racine. Par exemple : `/home/axel/Documents/Shell/TPs/TP1/Ex3/images/Chin`

4.

Le chemin relatif d'un fichier ou d'un repertoire correspond au chemin vers le fichier ou repertoire par rapport à un autre repertoire. Par exemple : `../P-Z/Vamporc.png`

6.

La commande `tar -xczf ITC313_TP_Shell_lebot.axel.tar.gz` permettra de décompresser et extraire l'archive.

7.

Après transfert de l'archive compressé toutes les permissions sont conservés.

1.1.4 Exercice 4 : Manipulation d'expression régulière

L'objectif de cet exercice est de manipuler les basiques des expressions régulière.

1.

La commande `wget https://cloud.infotro.fr/ITC313/unGrosBordel.txt` permettra de récupérer le fichier à analyser.

2.

Les lignes affichées par `cat unGrosBordel.txt | grep -E "ette"` contiennent toutes la suite de lettres "ette".

3.

Les lignes affichées par `cat unGrosBordel.txt | grep -E "T"` contiennent toutes la lettre "T".

4.
Les lignes affichées par `cat unGrosBordel.txt | grep -E "^T"` contiennent toutes la lettre "T" en début de ligne.

5.
L'expression ^ signifie donc "début".

6.
Les lignes affichées par `cat unGrosBordel.txt | grep -E "te$"` contiennent toutes la suite de lettres "te" en fin de ligne.

7.
Les lignes affichées par `cat unGrosBordel.txt | grep -E "c.r"` contiennent toutes la suite de lettres "c", un caractère quelconque, "r".

8.
Les lignes affichées par `cat unGrosBordel.txt | grep -E "(oui|non)"` contiennent toutes soit "oui", soit "non".

9.
- $ représante la fin d'une ligne.
- | représante une condition "ou".
- . représante un caractère quelconque.

10.
L'option `-o` dans la commande `cat unGrosBordel.txt | grep -o -E "c.r"` permet de n'afficher que la partie de la ligne correspondant à l'expression "c.r" appelé "motif".

11.
Les motifs affichées par `cat unGrosBordel.txt | grep -o -E "[A-Z]"` contiennent une suite de 4 lettres majuscule.

12.
Les motifs affichées par `cat unGrosBordel.txt | grep -o -E "[A-Z][a-z]+"` contiennent une suite d'au moins une majuscule et une minuscule.

13.
Les motifs affichées par `cat unGrosBordel.txt | grep -o -E "[A-Z][a-z]*"` contiennent une suite de 0, 1 ou plus de une majuscule et une minuscule.

14.
- + représante 1 ou plusieurs.
- * représante 0, 1 ou plusieurs.

15.
La commande

```
cat unGrosBordel.txt | grep -o -E "[A-Za-z0-9\.\_]+@([A-Za-z0-9\-]+\.)*[a-zA-Z]{2,4
```

permet de récupérer les addresse e-mail.

16.
La commande

```
cat unGrosBordel.txt | grep -o -E "(\+33|0)(\.| )?[0-9]((\.| )?[0-9]{2}){4}"
```

permet de récupérer les numéro de téléphone.

17.
La commande `cat unGrosBordel.txt | grep -o -E "\(\([a-z]+\)\)"` permet de trouver la phrase secrète : "bien joue tu as trouve la reponse a la derniere question"

1.2 Éditions de scripts

1.2.1 Exercice 5 : Un premier script

L'objectif de cet exercice est de créer un scripts basique et de l'éxécuter.

```
#!/bin/bash
# Axel LE BOT - 2017-10-01

# Récupération du nom de l'utilisateur
USER=$(whoami)

# On utilise la commande echo pour communiquer avec l'utilisateur
echo "Hello $USER !"
echo "Do you like fishsticks? (y/n)"

# On utilise la commande read pour récupère la réponse de l'utilisateur
read answer

# On vérifie si l'utilisateur a répondu oui
if [ "$answer" = "y" ]
then
        # Si il a répondu oui, on affiche le message suivant
        echo "Then, you are a gayfish!"
fi
```

1.2.2 Exercice 6 : Comptage des paramètres

Nous verrons ici comment le passage de paramètres, le comptage et la récupération des paramètres dans un script.

```
#!/bin/bash
# Axel LE BOT - 2017-10-02

echo "Liste des parametres entrés : "

# On utilie $* pour avoir tous les paramètres passés
for i in $*
do
```

```
        # On affiche le paramètre
        echo $i
done

# On utilise $# pour afficher le nombre de paramètres
echo "Nombre de parametres : $#"
```

La commande `shift` permet de décaler la liste d'arguments. Nous l'utiliserons dans le script suivant.

```
#!/bin/bash
# Axel LE BOT - 2017-10-02

echo "Liste des parametres entres : "

# On récupère le nombre de paramètres total
COUNT=$#

# Tant que shift décale
while shift
        # On affiche le paramètre
        echo $1
done

# Affichage du nombre de paramètre total
echo "Nombre de parametres : $COUNT"
```

1.2.3 Exercice 7 : Portée des variables

L'exercice suivant permettra de comprendre la portée de variable et de modifications entre shell.

1. Portée des variables locales

Après avoir créé initialisé une variables midichloriens à l'aide de la commande `midichloriens=50000`, om peut bien l'afficher avec la commande `echo midichloriens`.

```
# !/bin/bash
# Axel LE BOT - 2017-10-02
echo "Yoda"
echo "la valeur de midichloriens est $midichloriens"
```

On execute ensuite le script `yoda.sh` ci-dessus qui n'affiche rien. On en conclue donc que la variable initialisé dans le terminal n'est pas accessible depuis le script.

On essaie ensuite d'initialisé la variable `midichloriens` depuis le script `vador.sh` ci-dessous puis de ré-executer le script `yoda.sh`.

```
# !/bin/bash
# Axel LE BOT - 2017-10-02
echo "Vador"
```

```
midichloriens=20000
echo "la valeur de midichloriens est $midichloriens"
```

En éxécutant la commande `echo midichloriens` la variable `midichloriens` reste égale à 50000. On peut donc conclure que variables sont locales à un shell ou à un script.

2. Portée limitée au shell

En utilisant un second terminal, lorsque l'on affiche la valeur de la variable `midichloriens` en utilisant `echo midichloriens`, on constate que rien ne s'affiche.

En lancant un sous-shell avec la commande `bash` et en affichant la valeur de la variable `midichloriens`, rien ne s'affiche.

Après être sorti du sous-shell, on peut afficher la variable `midichloriens`, la valeur est toujours égale à 50000.

3. Étendre la portée de la valeur d'une variable locale

En ayant éxécuté la commande `export midichloriens` et refait les manipulations de 2., cette fois, la variable `midichloriens` est lisible par le sous-shell.

On initialise cette fois ci une variable `force` à 100. On modifie depuis le sous-shell cette valeur à 150. En fermant le sous-shell puis et en affichant la variable avec la commande `echo force`, on s'appercoit que la variable n'a pas été modifié.

La commande `export` permet d'exporter le clone d'une variable ayant le même comportement, le même nom, la même valeur.

II

C++

2. TP1+TP2 : Tableaux, matrices et Fonctio

2.1 Exercice sur des tableaux

2.1.1 Fonction sur les tableaux non triés

Exercice 1 : Algorithmes de parcours classiques sur tableau non triés

Quelques copier coller ont suffit. Les tests ont bien été éffectué.

Exercice 2 : Ajout et suppression d'éléments tableaux non triés

2.1.2 Algorithmes de tri de tableaux

Exercice 3 : Trier des tableaux aléatoires

2.1.3 Fonctions sur les tableaux triés

Exercice 4 : Algorithmes de parcours classiques sur tableau non triés

Exercice 5 : Ajout et suppression d'éléments sur tableaux triés

2.2 Fonctions récursives

L'objectif ce cette partie est d'utiliser les fonctions qui s'appelle elle-même appellé "récursives" et dans comprendre leur implémentation.

2.2.1 Exercice 6 : Definition de fonction recursive

On verra ici l'intéret d'utiliser des fonctions récursives sur des séries mathématique

Une fonction récursive est une fonction qui s'appelle elle-même. Si dans le corps de la fonction, nous l'utilisons elle-même, alors elle est récursive.

L'algorithme suivant permet de calculer la formation de Leibniz :

— Par iteration :

```
7  double PiLeibniz(int n) {
8      int i;
9      double result = 4 / 1;
```

```
10      for (i = 1; i <= n; i++) {
11          result = result + (pow(-1, i)) / ((2 * i) + 1);
12      }
13      return result;
14  }
```

— Par récursivité :

```
16  double PiLeibnizRecursif(int n) {
17      double result;
18      return (n == 1) ? 1 : PiLeibniz(n - 1) + (pow(-1, n)) / ((2 * n) + 1);
19  }
```

L'algorithme suivant permet de calculer un produit factorielle :

— Par iteration :

```
7   int factorielle(int n) {
8       int i;
9       int result = 1;
10      for (i = 1; i <= n; i++) {
11          result = result * i;
12      }
13      return result;
14  }
```

— Par récursivité :

```
16  int factorielleRecursif(int n) {
17      int result;
18      if (n == 1) result = 1;
19      else result = factorielleRecursif(n - 1) * n;
20      return result;
21  }
```

L'algorithme suivant permet de calculer la valeur d'un nombre harmonique :

— Par iteration :

```
7   double harmonique(int n) {
8       int i;
9       int result = 1;
10      for (i = 1; i <= n; i++) {
11          result = result + (1 / i);
12      }
13      return result;
14  }
```

— Par récursivité :

```
16  double harmoniqueRecursif(int n) {
17      int result;
18      if (n == 1) result = 1;
19      else result = harmoniqueRecursif(n - 1) + (1 / n);
```

```
20        return result;
21    }
```

2.2.2 Exercice 7 : Algorithme recursif sur matrice

Dans cet exercice nous implémenterons un algorithme récursif appliqué sur une matrice. On verra ainsi que la récursivité permet de solutionner beaucoup plus simplement un problème par rapport à un algorithme séquentiel.

Notre algorithme devra affecter la valeur 2 à chacune des cases de la case selectionné. Si la case est déjà égale à 2, rien ne sera fait. Les autres cases ne seront pas modifiés. Pour cela nous appelerons la fonction sur les cases de la même zone de la case selectionné.

Afin de rendre notre algorithme réutilisable nous devront passer en paramètre les dimensions de la matrice utilisé avant de passer la matrice en paramètre, ainsi la fonction connaitra les dimensions de la matrice.

Nous avons créé la fonction récursive suivante :

```
8  void remplir(int taille1, int taille2, int M[taille1][taille2], int i, int j) {
9      if (M[i][j] != 2) {
10         int tmp = M[i][j];
11         M[i][j] = 2;
12         if (i > 0 && M[i - 1][j] == tmp) {
13             remplir(taille1, taille2, M, i - 1, j);
14         }
15         if (i < taille1 - 2 && M[i + 1][j] == tmp) {
16             remplir(taille1, taille2, M, i + 1, j);
17         }
18         if (j > 0 && M[i][j - 1] == tmp) {
19             remplir(taille1, taille2, M, i, j - 1);
20         }
21         if (j < taille2 - 2 && M[i][j + 1] == tmp) {
22             remplir(taille1, taille2, M, i, j + 1);
23         }
24     }
25 }
```

3. TP3+TP4 : Manipulation des arbres

4. TP5+TP6 : Hiérarchie de processus, signa

5. TP7+TP8 : Communication socket

6. TP9+TP10 : Héritage multiple et modélisa

Ce TP nous permet de manipuler de manière concrète des mécanismes caractéristiques de la programmation orientée objet tels que la notion de classe, d'objet, d'héritage ainsi que l'héritage multiple, qui est propre au langage de programmation C++. Pour voir ces différentes notions nous allons créer un jeu simple.

Afin de structurer un projet et avoir un code aussi clair que possible en c++, il est conseillé de séparer son code en deux types de fichiers : les fichiers d'en-têtes `header (.h)` servant à définir les classes et donnant les différents prototype de méthodes, et les fichiers avec l'extension `.cpp`, que l'on devra compiler et qui contiennent tout le code des méthodes. On créé également un fichier main.cpp , que l'on doit compiler et qui contiendra la logique du jeu. Les fichiers `.h` sont inclus dans les fichiers `.cpp` correspondants ainsi que dans le main afin que l'on puisse avoir accès à la déclaration des classes et leurs prototypes de méthodes. Les fichiers .cpp sont compilés ensemble avec le compilateur g++ en un seul fichier exécutable.

Le jeu est un affrontement entre des peronnages, pouvant être de différents types : Guerrier, Mage ou Mage-Guerrier. Afin d'organiser ses idées et de bien commencer une programmation orientée object, on propose de modéliser les classes à l'aide du diagramme de classe suivant : objet

En premier nous allons définir la classe `Personne`, qui permet de définir les attributs communs à tous les combattants : le nom, de type `string` (chaîne de caractère), les points de vie et les points de défense, de type `int`. Ces attributs auront le spécificateur d'accès `protected`. Ce spécificateur précise que de sattributs sont accessibles au sein de la classe ainsi que ses classes filles. Nous allons donc créer des accesseurs en lecture afin de pouvoir lire les attributs des instances de la classe `Personne` dans le main si nécessaire avec des fonctions méthodes tels que :

```
33 int Personne::getPointsVie() {
34     return pointsVie;
```

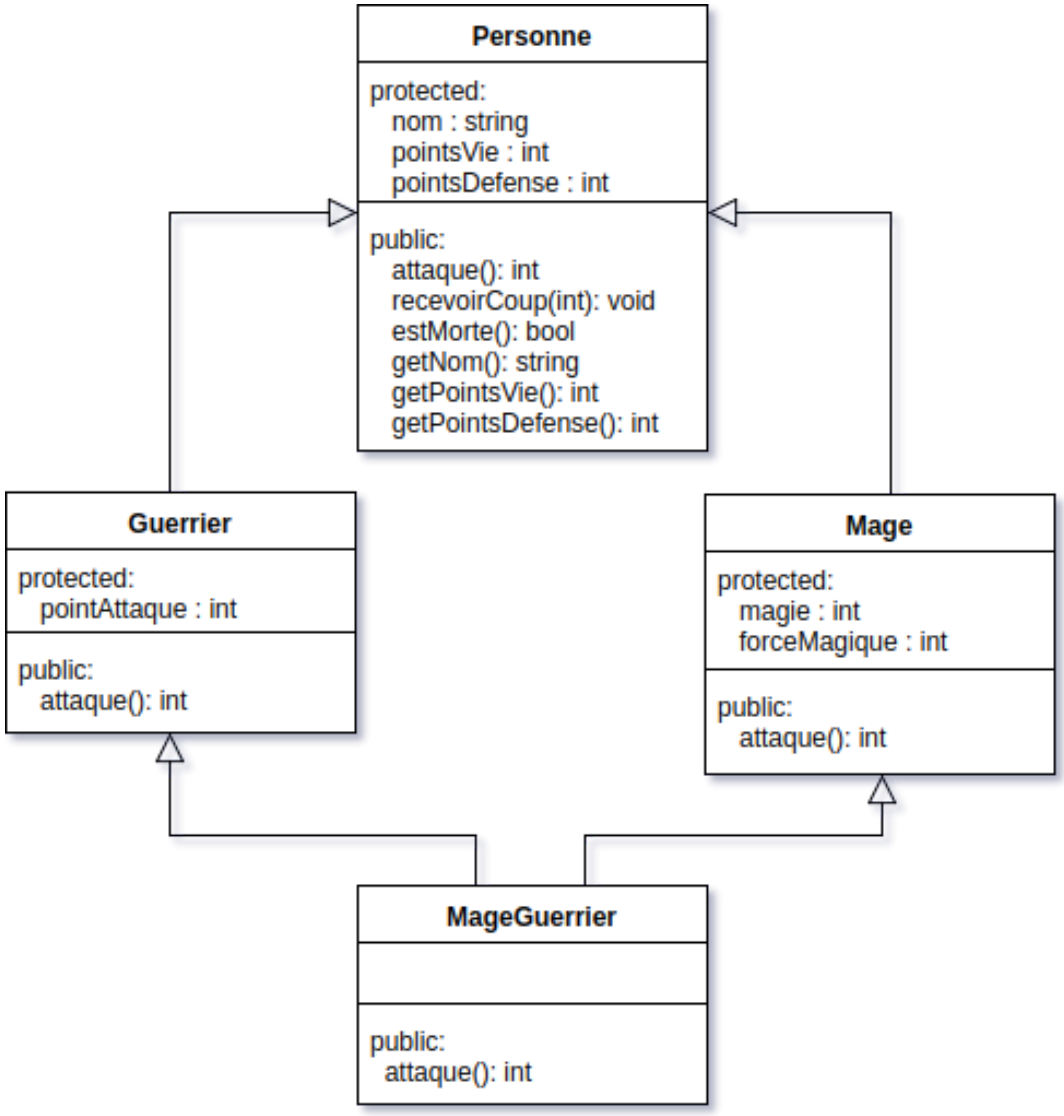

FIGURE 6.1 – Diagramme de classe

```
35 }
```

Afin d'initialiser correctement les membres de l'objet, on peut créer un constructeur et lui passer les valeurs que nous voulons attribués à l'initialisation de l'objet.

```
9  Personne::Personne(string nom, int pointsVie, int pointsDefense) {
10     this->nom = nom;
11     this->pointsVie = pointsVie;
12     this->pointsDefense = pointsDefense;
13 }
```

On ajoute de plus la méthode `recevoirCoup()` afin de permettre à une personne de recevoir un coup et faire baisser ses points de vie en fonction de ses points de défense. Nous pouvons directement accéder un attribut depuis la classe en écrivant le nom de la variable, nous pouvons aussi préciser qu'il s'agit de l'attribut de l'instance actuelle avec `this`.

```
}

bool Personne::estMorte() {
    return (pointsVie <= 0);
}
```

On ajoute la méthode `attaque()` afin que chaque personne puisse attaquer. Au niveau de la classe `Personne`, cette méthode est définit comme virtuel pure et ne sera pas définit et devra donc être redéfinit dans es classes filles. La classe `Personne` n'est plus instanciable.

```
class Personne {
protected:
    string nom;
    int pointsVie;
    int pointsDefense;

public:
    Personne() = delete;

    Personne(string nom, int pointsVie, int pointsDefense);

    virtual ~Personne();

    virtual int attaque()=0; //pure specifier

    void recevoirCoup(int coup);

    bool estMorte();

    string getNom();

    int getPointsVie();

    int getPointsDefense();
};
```

Comme dit précédemment les peronnes peuvent être, de différents type dans ce jeu. Nous allons donc créé deux classes : `Guerrier` et texttttMage héritant de la classe `Personne` (héritage simple). Ces classes auront accès aux même attributs mais auront en plus leur propre attributs et devront redéfinir la méthode `attaque()` et implémenteront leur propre logique. Ces deux classes auront aussi un constructeur pour initiliser leur propre attribut et devront appeler le constructeur de leur classe mère `Personne`.

La classe Guerrier :

```
class Guerrier : public virtual Personne {
protected:
    int pointsAttaque;
```

```
15 public :
16     Guerrier() = delete;
17
18     Guerrier(string nom, int pointsVie, int pointsDefense, int pointAttaque);
19
20     ~Guerrier();
21
22     int attaque() override;
23 };
```

La classe Mage :

```
12 class Mage : public virtual Personne {
13 protected:
14     int magie;
15     int forceMagique;
16
17 public:
18     Mage() = delete;
19
20     Mage(string nom, int pointsVie, int pointsDefense, int magie, int forceMagique)
21
22     ~Mage();
23
24     int attaque() override;
25
26 };
```

Cet exercice nous propose d'hériter de deux classe, celà s'appel l'héritage multiple qui est un mécanisme de programmation orientée objet dans lequel une classe peut hériter de comportements et de fonctionnalités de plus d'une super-classe. Ainsi nous pouvons créer un troisieme type de personnage `MageGuerrier` qui héritera à la fois de la classe `Mage` et de la classe `Guerrier`, bien entendu il héritera aussi inderectement de la classe Personne. Le constructeur devra comme pour les classes précédentes appeler les constructeur parents, donc ici les constructeur des classes `Personne`, `Guerrier` et `Guerrier`. Comme demander dans l'exercice on redéfinira la méthode `attaque()` qui devra attaquer "tantôt avec la magie, tantôt avec une attaque physique" (en tant que `Mage` ou en tant que `Guerrier`). On peut grâce à l'opérateur `::` appeler la méthode de notre choix, par exemple `Mage::attaque()` utilisera la méthode `attaque()` de la classe `Mage`.

La classe MageGuerrier :

```
7  MageGuerrier::MageGuerrier(string name, int pointsVie, int pointsDefense, int point
8                             int forceMagique) :
9          Personne(name, pointsVie, pointsDefense),
10         Mage(name, pointsVie, pointsDefense, magie, forceMagique),
11         Guerrier(name, pointsVie, pointsDefense, pointsAttaque) {
12 }
13
```

```
14 MageGuerrier::~MageGuerrier() {
15     cout << "MageGuerrier is being deleted\n";
16 }
17
18 int MageGuerrier::attaque() {
19     srand(time(NULL));
20     return rand() % 2 ? Mage::attaque() : Guerrier::attaque();;
21 }
```

Afin de tester nos classe et simuler un jeu nous avons implementer un combat dans le fichier `main.cpp`. Nous utiliserons la fonction `combat()` afin de simuler un combat entre deux personnes, pour cela nous devrons passer en paramètre les deux protagonistes. Afin de pouvoir passer n'importe quel type de personnage (`Guerrier`, `Mage` et `MageGuerrier`) il sera important de préciser le type des arguments à `Personne *` ce qui donne `combat(Personne * p1, Personne *p2)`. On applique ici le mécanisme de polymorphisme d'héritage qui permet de faire abstraction des détails des classes spécialisées d'une famille d'objet, en les masquant par une interface commune (qui est la classe de base).

Afin d'eviter les fuites mémoire à la fin de l'éxécution du programme il faut `delete` les objets créés.

```
43 int main() {
44     displayHeader(1, (char *) "TP 9-10");
45     bool finished = false;
46
47     Guerrier *g1 = new Guerrier("Guerrier 1", 50, 5, 15);
48     Mage *m1 = new Mage("Mage 1", 100, 3, 30, 12);
49     MageGuerrier *mg = new MageGuerrier("MageGuerrier 1", 100, 4, 20, 10, 25);
50
51     while (!finished) {
52         combat(m1, mg);
53         finished = endOfProgram();
54     }
55
56     delete g1;
57     delete m1;
58     delete mg;
59
60     printf("Bye !!!");
61     exit(0);
62 }
```