Informatique

Travaux Pratiques

Axel LE BOT

Table des matières

Table des figures

I 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

Dans cette partie on verra la différence de complexités de manipulation de tableau entre les tableaux qui sont triés et ceux qui sont non triés.

2.1.1 Fonction sur les tableaux non triés

Exercice 1 : Algorithmes de parcours classiques sur tableau non triés

Cet exercice mettra en avant les différents algorithmes de manipulation de tableaux non triés.

J'ai écrit une fonction `sommeElementsTab()` qui prend en paramètres un tableau d'entiers tab et sa taille taille, et renvoie la somme des éléments contenus dans le tableau.

```
7  int sommeElementsTab(int tab[], int taille) {
8      int somme = 0;
9      int i = 0;
10     for (i = 0; i < taille; i++) {
11         somme += tab[i];
12     }
13     return somme;
14 }
```

J'ai écrit une fonction `moyenneValeursTab()` qui prend en paramètres un tableau d'entiers tab et sa taille taille, et renvoie la moyenne des valeurs contenues dans le tableau.

```
16 int moyenneValeursTab(int tab[], int taille) {
17     return sommeElementsTab(tab, taille) / taille;
18 }
```

J'ai écrit une fonction `valeurContenueDansTabNonTrie()` qui prend en paramètres un tableau d'entiers tab et sa taille taille, ainsi qu'une valeur val, et renvoie 1 si le tableau contient au moins une occurence de la valeur val, et 0 sinon.

```
20 int valeurContenueDansTabNonTrie(int tab[], int taille, int val) {
21     int contenue = 0;
22     int i = 0;
23     while (i < taille && contenue == 0) {
24         if (tab[i] == val) contenue = 1;
25         i++;
26     }
27     return contenue;
28 }
```

Pour finir cet exercice, j'ai écrit une fonction `nbOccurencesValeurDansTabNonTrie()` qui prend en paramètres un tableau d'entierstab et sa taille taille, ainsi qu'une valeur val, et renvoie le nombre de fois que la valeur val est contenue dans le tableau.

```
30 int nbOccurencesValeurDansTabNonTrie(int tab[], int taille, int val) {
31     int occurence = 0;
32     int i = 0;
33     for (i = 0; i < taille; i++) {
34         if (tab[i] == val) occurence++;
35     }
36     return occurence;
37 }
```

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

Cet exercice mettra en avant les différents algorithmes de modification de tableaux non triés.

Nous implémenterons ici les fonctions pouvant Être utiles pour modifier des tableaux non triés.

J'ai écrit une fonction `ajoutValeurTabNonTrie()` qui prend en paramètres un tableau d'entier tab, sa taille maximum tailleMax, le nombre réel d'éléments qu'il contient taille, et une valeur à ajouter val. Cette fonction essaye d'ajouter `val` au tableau tab (par exemple à la fin du tableau), et retourne 1 si l'ajout a pu être correctement réalisé, et 0 sinon.

```
7  int ajoutValeurTabNonTrie(int tab[], int taille, int tailleMax, int val) {
8      if (tailleMax == taille) return 0;
9      else {
10         tab[taille] = val;
11         return 1;
12     }
13 }
```

J'ai écrit une fonction `supprimeValeurTabNonTrie()` qui prend en paramètres un tableau d'entier tab, le nombre réel d'éléments qu'il contient taille, et une valeur à supprimer val. Cette fonction essaye de supprimer la première occurence de val rencontrée dans le tableau tab, et retourne 1 si la suppression a pu être correctement réalisée (le tableau

contenait cette valeur), et 0 sinon. Pour supprimer une telle valeur dans un tableau non trié, une astuce consiste à chercher l'indice de la première cellule contenant la valeur à supprimer, à inverser cette valeur avec la valeur de la dernière cellule, et considérer que la taille du tableau sera diminuée de 1. L'inversion des valeurs de deux cellules n'a pas d'incidence ici puisque le tableau n'est pas censé être trié.

```
15 int supprimeValeurTabNonTrie(int tab[], int taille, int val) {
16     int i = 0, suppression = 0, temp;
17     while (i < taille && suppression == 0) {
18         if (tab[i] == val) {
19             temp = tab[i];
20             tab[i] = tab[taille - 1];
21             tab[taille - 1] = temp;
22             suppression = 1;
23         }
24         i++;
25     }
26     return suppression;
27 }
```

J'ai écrit une fonction `supprimeToutesLesValeursTabNonTrie()` qui prend en paramètres un tableau d'entier tab, le nombre réel d'éléments qu'il contient taille, et une valeur à supprimer val. Cette fonction essaye de supprimer toutes les occurences de val rencontrée dans le tableau tab, et retourne le nombre d'éléments qui ont été supprimés et 0 sinon.

```
29 int supprimeToutesLesValeursTabNonTrie(int tab[], int taille, int val) {
30     int compteur = 0;
31     while (supprimeValeurTabNonTrie(tab, taille - compteur, val)) {
32         compteur++;
33     }
34     return compteur;
35 }
```

2.1.2 Algorithmes de tri de tableaux

Exercice 3 : Trier des tableaux aléatoires

Cet exercice mettra en avant les algorithmes permettant de trier un tableau, ce qui nous sera utile pour améliorer les fonctions précédemment implémentés.

Nous avons défini les fonctions précédentes pour le cas d'un tableau aléatoire, non trié. Il existe plusieurs algorithmes permettant de trier un tableau, ici nous allons utiliser deux de ces algorithmes et les implémenter afin de l'utiliser sur nos tableaux non triés.

J'ai écrit une fonction `triABulle()` qui prend en paramètre un tableau d'entiers et sa taille (effective), et trie les éléments du tableau selon l'algorithme de tri tri à bulle. Le principe du tri à bulles est de comparer deux valeurs adjacentes et d'inverser leur position si elles sont mal placées.

```
7  void triABulle(int tab[], int taille) {
8      int i, j, tmp;
```

```
9       for (i = taille - 1; i > 0; i--) {
10          for (j = 0; j < i; j++) {
11              if (tab[j + 1] < tab[j]) {
12                  tmp = tab[j];
13                  tab[j] = tab[j + 1];
14                  tab[j + 1] = tmp;
15              }
16          }
17      }
18  }
```

J'ai écrit une fonction `triSelection()` qui prend en paramètre un tableau d'entiers et sa taille (effective), et trie les éléments du tableau selon l'algorithme de tri tri par sélection. Le principe de cet algorithme est rappelé ci-après. Il vous revient de définir les variables intermédiaires au bon endroit, ou d'adapter les valeurs en fonction du cas de figure. L'idée est simple : rechercher le plus grand élément (ou le plus petit), le placer en fin de tableau (ou en début), recommencer avec le second plus grand (ou le second plus petit), le placer en avant-dernière position (ou en seconde position) et ainsi de suite jusqu'à avoir parcouru la totalité du tableau.

```
20  void triSelection(int tab[], int taille) {
21      int i, j, tmp;
22      for (i = 0; i < taille - 1; i++) {
23          for (j = i; j < taille; j++) {
24              if (tab[j] < tab[i]) {
25                  tmp = tab[j];
26                  tab[j] = tab[i];
27                  tab[i] = tmp;
28              }
29          }
30      }
31  }
```

Nous pouvons maintenant trier les tableaux ce qui nous permet de travailler sur les tableaux triés.

2.1.3 Fonctions sur les tableaux triés

Exercice 4 : Algorithmes de parcours classiques sur tableau triés

J'ai écrit une fonction `valeurContenueDansTabTrie()` qui prend en paramètres un tableau d'entiers tab et sa taille taille, ainsi qu'une valeur val, et renvoie 1 si le tableau contient au moins une occurence de la valeur val, et 0 sinon. L'algorithme modifié fonctionne par recherche dichotomique, comme vu en cours : on définit les indices min et max de recherche de la valeur dans le tableau (respectivement initialisés à 0 et taille), et on regarde la valeur contenue dans la case d'indice médian ((min + max) /2). Selon que la valeur lu est plus grande, plus petite ou égale, on fait varier la valeur de l'indice min, ou max, ou on conclut.

```
7  int valeurContenueDansTabTrie(int tab[], int taille, int val) {
8      int iMin = 0;
9      int iMax = taille - 1;
10     int iMid;
11     int trouve = -1;
12     while ((iMin <= iMax) && (trouve == -1)) {
13         iMid = (iMax + iMin) / 2;
14         if (val < tab[iMid])
15             iMax = iMid - 1;
16         else if (val > tab[iMid])
17             iMin = iMid + 1;
18         else
19             trouve = 1;
20     }
21     return (trouve);
22 }
```

J'ai écrit une fonction `nbOccurencesValeurDansTabTrie()` qui prend en paramètres un tableau d'entierstab et sa taille taille, ainsi qu'une valeur val, et renvoie le nombre de fois que la valeur val est contenue dans le tableau. La version optimisée consiste à rechercher une valeur, comme vu précédemment. Si cette valeur existe, on regarde simplement le nombre de cellules à gauche et à droite contenant cette valeur.

```
24 int nbOccurencesValeurDansTabTrie(int tab[], int taille, int val) {
25     int iMax = taille - 1, iMin = 0;
26     int iMid = (iMin + iMax) / 2;
27     int compteur = 0;
28     int i;
29     while ((iMin <= iMax) && (tab[iMid] != val)) {
30         if (tab[iMid] > val) {
31             iMax = iMid - 1;
32         } else {
33             iMin = iMid + 1;
34         }
35         iMid = (iMin + iMax) / 2;
36     }
37     i = iMid;
38     while ((i >= iMin) && (tab[i] == val)) {
39         ++compteur;
40         --i;
41     }
42     i = iMid + 1;
43     while ((i <= iMax) && (tab[i] == val)) {
44         ++compteur;
45         ++i;
46     }
47     return compteur;
48 }
```

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

J'ai écrit une fonction `ajoutValeurTabTrie()` qui prend en paramètres un tableau d'entier tab trié, sa taille maximum tailleMax, le nombre réel d'éléments qu'il contient taille, et une valeur à ajouter val. Cette fonction essaye d'ajouter val au tableau tab en respectant le tri (par exemple à la fin du tableau), et retourne 1 si l'ajout a pu être correctement réalisé, et 0 sinon. L'algorithme consiste à décaler les valeurs de 1 à droite en partant de la fin, jusqu'à trouver l'endroit où insérer la valeur.

```
7  int ajoutValeurTabTrie(int tab[], int taille, int tailleMax, int val) {
8      int i;
9      if (taille != tailleMax) {
10         i = taille - 1;
11         if (val > tab[i]) {
12             tab[taille] = val;
13
14         } else {
15             while ((val < tab[i]) && (i >= 0)) {
16                 tab[i + 1] = tab[i];
17                 i--;
18             }
19             tab[i + 1] = val;
20         }
21         return 1;
22     } else {
23         return 0;
24     }
25 }
```

J'ai écrit une fonction `supprimeValeurTabTrie()` qui prend en paramètres un tableau d'entier tab trié, le nombre réel d'éléments qu'il contient taille, et une valeur à supprimer val. Cette fonction essaye de supprimer la première occurence de val rencontrée dans le tableau tab, et retourne 1 si la suppression a pu être correctement réalisée (le tableau contenait cette valeur), et 0 sinon. Pour supprimer une telle valeur dans un tableau trié, une astuce consiste à chercher l'indice de la première cellule contenant la valeur à supprimer (par exemple en utilisant la recherche dichotomique), puis à décaler les valeurs successives de un vers la gauche jusqu'à la fin du tableau.

```
27 int supprimeValeurTabTrie(int tab[], int taille, int val) {
28     int iMax = taille - 1, iMin = 0;
29     int iMid = (iMin + iMax) / 2;
30     while ((iMin != iMax) && (tab[iMid] != val)) {
31         if (tab[iMid] > val) {
32             iMax = iMid - 1;
33         } else {
34             iMin = iMid + 1;
35         }
36         iMid = (iMin + iMax) / 2;
37     }
```

```
38      if (tab[iMid] != val) {
39          return 0;
40      } else {
41          for (int i = iMid; i < taille - 1; ++i) {
42              tab[i] = tab[i + 1];
43          }
44          return 1;
45      }
46  }
```

J'ai écrit une fonction `supprimeToutesLesValeursTabTrie()` qui prend en paramètres un tableau d'entier tab trié, le nombre réel d'éléments qu'il contient taille, et une valeur à supprimer val. Cette fonction essaye de supprimer toutes les occurences de val rencontrée dans le tableau tab, et retourne le nombre d'éléments qui ont été supprimés et 0 sinon. Cet algorithme est le moins facile : il consiste à trouver une occurence au moyen d'une recherche dichotomique, puis se positionner sur celle la plus à gauche, compter le nombre d'occurences de cette valeur (contenues dans les cellules immédiatement suivantes), et décaler les valeurs suivantes du nombre d'occurences calculé.

```
48  int supprimeToutesLesValeursTabTrie(int tab[], int taille, int val) {
49      int iMax = taille - 1, iMin = 0;
50      int iMid = (iMin + iMax) / 2;
51      int compteur = 0;
52      int i;
53      int iGauche;
54      while ((iMin != iMax) && (tab[iMid] != val)) {
55          if (tab[iMid] > val) {
56              iMax = iMid - 1;
57          } else {
58              iMin = iMid + 1;
59          }
60          iMid = (iMin + iMax) / 2;
61      }
62      if (tab[iMid] != val) {
63          return 0;
64      } else {
65          i = iMid;
66          while ((i >= iMin) && (tab[i] == val)) {
67              --i;
68              ++compteur;
69          }
70          iGauche = i + 1;
71          i = iMid + 1;
72          while ((i <= iMax) && (tab[i] == val)) {
73              ++compteur;
74              ++i;
75          }
```

```
76          for (int i = iGauche; i < taille - 1; ++i) {
77              tab[i] = tab[i + compteur];
78          }
79          return compteur;
80      }
81  }
```

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

Un arbre binaire de recherche (ABR) est une structure de données pour représenter un ensemble ou un tableau associatif dont les clés appartiennent à un ensemble totalement ordonné. Un arbre binaire de recherche permet des opérations rapides pour rechercher une clé, insérer ou supprimer une clé. Un arbre binaire de recherche est un arbre binaire dans lequel chaque nœud possède une clé, telle que chaque nœud du sous-arbre gauche ait une clé inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous-arbre droit possède une clé supérieure ou égale à celle-ci — selon la mise en œuvre de l'ABR, on pourra interdire ou non des clés de valeur égale. Les nœuds que l'on ajoute deviennent des feuilles de l'arbre.

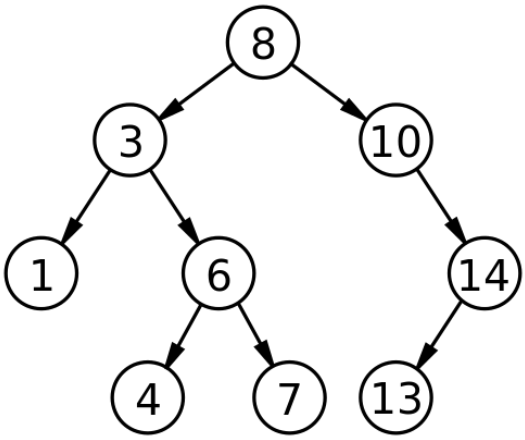

FIGURE 3.1 – Arbre Binaire de Recherche

3.1 Algorithmes sur arborescences binaires de recherche (ABR) non équilibrées

3.1.1 Exercice 1 : Mise en place d'ABR et premiers algorithmes

Cet exercice permettra de mettre en place les structures d'ABR (Arbres Binaires de Rechercher) et de créer les fonctions de calcules pour les arbres.

J'ai écrit une fonction `nouveauNoeud()` qui prend en paramètre une valeur, et réserve la mémoire nécessaire au stockage d'un nouveau nœud dans le tas. Elle affecte alors l'élément valeur avec la valeur passée en paramètre, et fait pointer filsGauche et filsDroit vers null. Enfin, elle retourne un pointeur vers le nœud créé.

```
9  struct noeud *nouveauNoeud(int val) {
10     struct noeud *noeudRetour;
11     // reservation d'un espace memoire dans le tas
12     noeudRetour = (struct noeud *) malloc(sizeof(struct noeud));
13
14     // affectation des valeurs si l'allocation memoire reussit
15     if (noeudRetour != NULL) {
16         noeudRetour->valeur = val;
17         noeudRetour->filsGauche = NULL;
18         noeudRetour->filsDroit = NULL;
19     } else {
20         printf("erreur : plus de memoire (gros probleme!)\n");
21         exit(1);
22     }
23     //renvoi du pointeur vers le noeud nouvellement cree
24     return noeudRetour;
```

J'ai écrit une fonction `estVide()` qui prend en paramètres un pointeur sur un noeud de l'arbre, et renvoie 1 si le pointeur a comme valeur NULL, et 0 sinon. Cette fonction permet par la suite de déterminer plus facilement si un sous-arbre est vide.

```
76 int estVide(struct noeud *noeudPointe) {
77     return (noeudPointe == NULL) ? 1 : 0;
78 }
```

J'ai écrit une fonction `estFeuille()` qui prend en paramètres un pointeur sur un noeud de l'arbre, et renvoie 1 si le noeud pointé est une feuille (le noeud existe et ses deux fils pointent sur NULL), et 0 sinon.

```
80 int estFeuille(struct noeud *noeudPointe) {
81     return (noeudPointe != NULL &&
82             noeudPointe->filsGauche == NULL &&
83             noeudPointe->filsDroit == NULL) ? 1 : 0;
84 }
```

J'ai écrit une fonction `rechercheValeur()` qui prend deux paramètres, le premier étant un pointeur sur la racine de l'arbre, et le second une valeur v à chercher. La fonction renvoie 1 si la valeur v est contenue dans l'arbre, et 0 sinon. Le déroulement de la fonction

est le suivant : un pointeur parcours pointe sur le premier élément de l'arbre. Si cet élément est vide (pointeur null), la valeur v n'existe pas, et on renvoie 0. si cet élément existe, on regarde sa valeur. Si cette valeur est la valeur recherchée, on renvoie 1. Sinon, on déplace le pointeur parcours vers le fils gauche ou le fils droit du noeud selon que la valeur recherchée strictement inférieure ou strictement supérieure à la valeur du noeud pointé. Et on recommence.

```
87  //renvoie 1 si la valeur val est contenue dans l'arbre, 0 sinon
88  int rechercheValeur(struct noeud *noeudPointe, int val) {
89      struct noeud *courant = noeudPointe;
90      int trouve = 0;
91      if (courant == NULL) {
92          trouve = 0;
93      } else if (courant->valeur > val) {
94          trouve = rechercheValeur(courant->filsGauche, val);
95      } else if (courant->valeur < val) {
96          trouve = rechercheValeur(courant->filsDroit, val);
97      } else {
98          trouve = 1;
99      }
100     return trouve;
101 }
```

3.1.2 Exercice 2 : Algorithmes récursifs sur arborescences

Cet exercice permettra de rajouter des algorithmes récursifs pour les arbres.

J'ai écrit une fonction `nbNoeuds()` qui prend en paramètre un pointeur noeudCourant vers le premier nœ ud d'un arbre, et renvoie son nombre de noeuds. Pour écrire cette fonction de manière récursive. Le nombre de noeuds d'un arbre vide (c.a.d. noeudCourant est égal à NULL) est égal à 0. Sinon, il est égal à 1 + le nombre de noeuds de son sous-arbre droit + le nombre de noeuds de son sous-arbre gauche.

```
16  int nbNoeuds(struct noeud *noeudCourant) {
17      int nb;
18      if (estVide(noeudCourant)) {
19          nb = 0;
20      } else {
21          nb = 1 + nbNoeuds(noeudCourant->filsGauche) + nbNoeuds(noeudCourant->filsDr
22      }
23
24      return nb;
25  }
```

J'ai écrit une fonction `sommeValArbres()` qui prend en paramètre un pointeur noeud-Courant vers le premier nœ ud d'un arbre, et renvoie la somme des valeurs des noeuds. Pour écrire cette fonction de manière récursive. La somme des noeuds d'un arbre vide (c.a.d. noeudCourant est égal à NULL) est égal à 0. Sinon, elle est égale à la valeur contenue dans le noeud + la somme des noeuds de son sous-arbre droit + la somme des noeuds de son sous-arbre gauche.

```
26
27 int sommeValArbres(struct noeud *noeudCourant) {
28     int somme;
29
30     if (estVide(noeudCourant)) {
31         somme = 0;
32     } else {
33         somme = noeudCourant->valeur + sommeValArbres(noeudCourant->filsGauche) +
34                 sommeValArbres(noeudCourant->filsDroit);
35     }
36
37     return somme;
38 }
```

J'ai écrit une fonction `hauteur()` qui prend en paramètre un pointeur noeudCourant vers le premier nœ ud d'un arbre, et renvoie sa hauteur. Pour écrire cette fonction de manière récursive. La hauteur d'un arbre vide (c.a.d. noeudCourant est égal à NULL) est égal à 0. Sinon, elle est égale au maximum des profondeurs de chacun de ses sous-arbres + 1

```
40 int hauteur(struct noeud *noeudCourant) {
41     int h;
42
43     if (estVide(noeudCourant)) {
44         h = 0;
45     } else {
46         h = 1 + max(hauteur(noeudCourant->filsGauche), hauteur(noeudCourant->filsDr
47     }
48
49     return h;
50 }
```

J'ai écrit une fonction `detruireArbre()` qui prend en paramètre un pointeur vers la racine de l'arbre racine, et détruit l'arbre en libérant sa mémoire. Avant de libérer la mémoire d'un noeud, il faut bien évidemment libérer la mémoire de chacun de ses sous-arbres.

```
52 void detruireArbre(struct noeud *noeudRacine) {
53     if (!estVide(noeudRacine)) {
54         detruireArbre(noeudRacine->filsGauche);
55         detruireArbre(noeudRacine->filsDroit);
56         free(noeudRacine);
57     }
58 }
```

3.2 Modification et parcours d'ABR non équilibrées

3.2.1 Exercice 3 : Insertion et suppression de valeurs dans une arborescence

Cet exercice permettra de rajouter les algorithmes permettant de modifier les arbres (supprimer et ajouter des noeuds).

J'ai écrit une fonction `ajouterValeurABR()` qui prend comme paramètres un pointeur racine vers le premier nœ ud d'un arbre ainsi qu'une valeur, et ajoute cette valeur à l'arbre. De plus, elle renvoie un pointeur vers le premier noeud de l'arbre, au cas où ce dernier aurait changé (ce qui est le cas si l'arbre était vide). Rappelons que l'on ajoute un noeud en tant que nouvelle feuille, à l'unique endroit possible pour maintenir le caractère de recherche de l'arbre binaire.

```
8  struct noeud *ajouterValeurABR(struct noeud *noeud, int val) {
9      if (noeud == NULL) {
10         noeud = nouveauNoeud(val);
11     } else {
12         if (noeud->valeur < val) {
13             noeud->filsDroit = ajouterValeurABR(noeud->filsDroit, val);
14         } else if (noeud->valeur > val) {
15             noeud->filsGauche = ajouterValeurABR(noeud->filsGauche, val);
16         }
17     }
18     return noeud;
19 }
```

J'ai écrit une fonction `supprimerValeurABR()` qui prend comme paramètres un pointeur racine vers le premier nœud d'un arbre ainsi qu'une valeur, et supprime cette valeur de l'arbre si cette dernière existait. De plus, elle renvoie un pointeur vers le premier noeud de l'arbre, au cas où ce dernier aurait changé (ce qui est le cas si le premier noeud était la valeur à supprimer). Rappelons que la suppression d'une valeur d'un noeud doit maintenir le caractère de recherche de l'arbre binaire.

```
21 struct noeud *supprimerValeurABR(struct noeud *racine, int val) {
22     struct noeud *courant; // pointe sur le noeud a l'etude
23     struct noeud *ainverser; // pointe sur le noeud a dont la valeur sera a inverse
24     struct noeud *precedent; // pointe vers noeud precedent celui a supprimer
25     struct noeud *ptrRetour; // pour retourner l'adresse de la racine de l'arbre
26     int temp;
27     char direction;  // a chaque fois que l'on avance, permet de savoir si on venai
28
29     // initialisation de parcours
30     precedent = NULL;
31     courant = racine;
32
33     // d'abord on repere si la valeur a supprimer existe
34     // on avance courant jusqu'a ce qu'il pointe sur le neoud contenant la valeur a
35     while (courant != NULL && courant->valeur != val) {
36         precedent = courant;
```

```
37          if (courant->valeur <= val) {
38              direction = 'g';
39              courant = courant->filsDroit;
40          } else {
41              courant = courant->filsGauche;
42              direction = 'd';
43          }
44      }
45      // si elle n'existe pas :
46      if (courant == NULL) {
47          printf("valeur inexistante\n");
48          return racine;
49      }
50
51      // cas particulier : la racine doit etre supprimee et n'a pas de fils
52      // l'arbre deviendra vide apres suppression
53
54      if (racine->valeur == val && estFeuille(racine)) {
55          free(racine);
56          racine = NULL;
57          return racine;
58      }
59
60      // algorithm general
61      // tant que le noeud a supprimer a des fils, on avance selon l'algorithme
62      while (!estFeuille(courant)) {
63
64          // si on peut aller a gauche : on avance une fois gauche puis tout a droite
65          // (on oublie pas qu'il nous faudra connaitre le parent du nouveau noeud)
66          if (courant->filsGauche != NULL) {
67              // on avance une fois a gauche
68              direction = 'd';
69              ainverser = courant->filsGauche;
70              precedent = courant;
71              // on avance tout a droite
72              while (ainverser->filsDroit != NULL) {
73                  direction = 'g';
74                  precedent = ainverser;
75                  ainverser = ainverser->filsDroit;
76              }
77              // on inverse les valeurs
78
79              temp = ainverser->valeur;
80              ainverser->valeur = courant->valeur;
81              courant->valeur = temp;
82
```

```
83              // on recommencera la boucle a partir du noeud contenant la valeur a su
84              courant = ainverser;
85          } else
86              // on ne pouvait pas aller a gauche. On doit pouvoir aller a droite
87          {           // on avance une fois a droite
88              direction = 'g';
89              ainverser = courant->filsDroit;
90              precedent = courant;
91              // on avance tout a gauche
92              while (ainverser->filsGauche != NULL) {
93                  direction = 'd';
94                  precedent = ainverser;
95                  ainverser = ainverser->filsGauche;
96              }
97              // on inverse les valeurs
98
99              temp = ainverser->valeur;
100             ainverser->valeur = courant->valeur;
101             courant->valeur = temp;
102
103             // on recommencera la boucle a partir du noeud contenant la valeur a su
104             courant = ainverser;
105         }
106     }
107
108     // on est au bout. Le noeud a supprimer est une feuille, et precedent doit poin
109     if (direction == 'g') {
110         courant = NULL;
111         free(courant);
112         printf("Precedent nul : %i\n", precedent == NULL);
113         precedent->filsDroit = NULL;
114     } else {
115         courant = NULL;
116         free(courant);
117         printf("Precedent nul : %i\n", precedent == NULL);
118         precedent->filsGauche = NULL;
119     }
120
121     // on retourne la racine
122     return racine;
123 }
```

3.3 Parcours d'arborescences binaires

3.3.1 Exercice 4 : Parcours sur arbres

Cet exercice permet de lister tous les éléments d'un arbre de manière récursive.

J'ai écrit une fonction `parcoursProfondeur()` permettant de réaliser un parcours en profondeur, c'est à dire lister les éléments dans l'ordre suivant : (sous-arbre gauche) element (sous-arbre droit).

```
8  void parcoursProfondeur(struct noeud *racine) {
9      // sinon, on affiche son sous-arbre gauche, puis la valeur du noeud
10     // puis son sous-arbre droit
11     // si le noeud racine est NULL, on ne fait rien
12     if (!estVide(racine)) {
13         if (racine->filsGauche != NULL) {
14             parcoursProfondeur(racine->filsGauche);
15         }
16         printf("%i", racine->valeur);
17         if (racine->filsDroit != NULL) {
18             parcoursProfondeur(racine->filsDroit);
19         }
20     }
21 }
```

4. TP5+TP6 : Hiérarchie de processus, signa

Dans ce TP, nous allons revenir sur quelques commandes en shell pour illustrer le fonctionnement de manière simple, puis nous déporterons ces concepts au travers d'appels systèmes dans une fonction C.

4.1 Gestion des signaux : envoi et reception

Les exercices de cette question se concentrent sur les mécanismes d'envoi et de reception des signaux. Ils visent prioritairement à comprendre comment on envoie un signal, quelles sont les conditions nécessaires pour qu'un processus accepte de recevoir un signal, et comment dérouter l'exécution normale d'un programme lorsque ce dernier reçoit un signal. Nous montrons notamment que le concept d'envoi et réception de signaux s'applique aussi bien pour des scripts shell que pour des programmes écrits dans un langage de programmation, ici le C. Dans la section d'après, nous verrons comment ces signaux sont utilisés pour gérer les processus, notamment les phases d'exécution et d'arrêt prématuré.

4.1.1 Exercice 1 : Droits et signaux

Cet exercice à pour objectif de nous faire comprendre la nation de droits associés aux signaux. Pour cela, nous utiliserons plusieurs commandes shell, notamment la commande `ps` qui permet de lister les processus actifs sur une machine, et la commande `kill` qui permet d'envoyer un signal à un processus identifié par son pid.

La commande `kill -l` nous permet de lister l'ensemble des signaux. Pour pouvoir les compter nous éxecutons la commande `kill -l | wc -w` qui compte 31 signaux sur ma machine. On peut envoyer un signal à un processus dont on connait son `PID` à l'aide de la commande `kill -<SIG_NAME> <PID>`. Par exempler nous pouvons tuer le processus 1285 à l'aide de la commande `kill -SIGKILL 1285`.

Nous écrivons un programme permettant de boucler à l'infinie à l'aide d'une boucle

while(1). Le programme était en BASH je l'ai implémenté en C :

```
//
// Created by Axel LE BOT on 08/11/17.
//

int main() {
    printf("Lancement du processus %i\n", getpid());
    while (true) {
        printf(".\n");
        sleep(1);
    }
}
```

En éxecutans ce programme dans et en récupérant son PID, nous pouvons dans un terminal, en utilisant la commande kill envoyer un signal SIGKILL à ce processus et ainsi mettre fin au programme en le tuant. Si j'envoie le signal à un processus dont je ne suis pas le propriétaire (appartenant à autre utilisateur ou au root), l'erreur "Operation not authorized" s'affiche. Je peut le vérifier en éxécutant la commande sur un processus root et en vérificant son éxécution dans la liste des processus à l'aide de la commande top.

4.1.2 Exercice 2 : capture de signal et traduction en langage C

Cet exercice nous apprendra capturer des signaux, c'est à dire que l'on peut détecter la réception d'un signal envoyé par un autre processus, et redéfinir le comportement à adopter, c'est à dire la suite d'instructions à exécuter, lorsqu'un signal est détecté.

J'ai écrit un script shell comportant une boucle infinie affichant un "." chqaue seconde. Le script capture aussi le signal SIGINT et SIGUSR1 ainsi si l'utilisateur appuie sur CTRL+C ou envoie un signal SIGUSR1 pendant son éxécution un message sera affiché. Ce script lancé ne pourra donc pas être arrété si l'utilisateur appuie sur CTRL+C ou utilise la commande killl -SIGKILL <PID>. Le script et le suivant :

```
#!/bin/bash
# Author : Axel LE BOT

echo "lancement du processus $$";

while :
do
    sleep 1;
    echo ".";
    trap "echo SIGNAL CAPTURE; echo FIN" SIGINT
    trap "echo SIGNAL USER01 ; echo FIN" SIGUSR1
done
```

Et maintenant en langage C :

```
void signalsHandler(int interruption) {
    printf("Signal received\n");
```

```
7       switch (interruption) {
8           case SIGINT:
9               printf("SIGNAL SIGINT\n");
10              break;
11          case SIGUSR1 :
12              printf("SIGNAL SIGUSR1\n");
13              break;
14          default:
15              printf("SIGNAL NOT FOUND\n");
16      }
17  }
18
19  int main() {
20      signal(SIGINT, signalsHandler);
21      signal(SIGUSR1, signalsHandler);
22
23      while (true) {
24          sleep(1);
25          printf(".\n");
26      }
27  }
```

4.1.3 Exercice 3 : Capture de signaux et redirections (exercice difficile)

Cet exercice permet de capturés des signaux et les redirigés, c'est à dire que lorsque le signal est reçu, on redéfinit le comportement par défaut qu'aurait du avoir le processus. Dans un premier temps, nous allons voir quel est le comportement par défaut des processus lors de la réception d'un signal. Dans un second temps, nous allons redéfinir le comportement d'un processus à la réception d'un signal.

Plusieurs signaux permettent de terminer l'exécution d'un porocessus. Afin de pouvoir savoir lesquels nous allons écrire un script. Pour chaque signal X, un processus sera lancé en arrière plan, ici le script de la boucle infinie sera utilisé. Son PID sera retrouvé grace à la variable `$!` afin de lui envoyer le signal X. À l'aide de la commande `ps -p <PID>` nous saurons si le processus à été arrété, ainsi on pourra conclure que le signal X tuje ou pas le processus. Voici le script :

```
1   #!/bin/bash
2   # Author : Axel LE BOT
3
4   for signal in "seq 1 31"
5   do
6       # On lance le script de la boucle infinie en arrière plan et en redirigeant la
7       ./boucleShell.sh & > /dev/null 2>&1
8       # On récupère l'id du processus
9       PID=$!
10      echo "Process $PID"
11
12      # On tue le processus avec le signal
```

```
    kill -$signal $PID > /dev/null 2>&1

    # On vérifie si le processus est en cours d'execution
    if ps -p $PID > /dev/null 2>&1
    then
        # Si il est toujours en cours d'execution, on le tue.
        echo "Process not ended"
        kill -9 $PID > /dev/null 2>&1
    else
        # Sinon on affiche le signal qui à réussi à tuer
        echo "Process ended with the signal : $signal"
    fi
    echo "------"
    # Attendre
    sleep 0
done
```

4.1.4 Exercice 4 : envoi multiples et capture de signal en C

Au terme des exercices précédents nous avons compris les notions de signal, de redirection et de mise en oeuvre en langage shell. L'objectif de cet exercice est la simple traduction de ce qui a été vu en shell en langage C, et l'étude d'une autre commande nommée killall Nous avons besoin dans un premier temps du code ayant une boucle infinie et capturant le signal SIGINT (CTRL+C). Le script est ci-dessous :

```
#!/bin/bash
# Author : Axel LE BOT

trap "echo Signal CTRL-C capturé" SIGINT
echo "lancement du processus $$";
while :
do
    sleep 1;
done
```

Et maintenant en langage C :

```
void signalsHandler(int interruption) {
    printf("Signal received\n");
    switch (interruption) {
        case SIGINT:
            printf("SIGNAL SIGINT\n");
            break;
        default:
            printf("SIGNAL NOT FOUND\n");
    }
}

```

```
16 int main() {
17     printf("Lancement du processus %i\n", getpid());
18     signal(SIGINT, signalsHandler);
19     while (true) {
20         printf(".\n");
21         sleep(1);
22     }
23 }
```

Nous pouvons maintenant lancer 3 processus de notre programme C afin de les tuers avec la commande `killall -SIGINT ex4`. On essaie aussi de les mettre en pause et des les relancer grace aux commandes

— `killall -SIGSTOP ex4`
— `killall -SIGCONT ex4`

Puis nous les tuons grace à la commande `killall -9 ex4`

4.2 Gestion des processus

4.2.1 Exercice 5 : Processus en premier-plan / Arriere-plan

Nous allons voir ici comment manipuler les processus pour les passer tantôt en premier-plan, tantôt en arrière-plan, les stopper et les relancer. Ces manipulations sont rendues possibles par l'utilisation de signaux, et les fonctionnalités de deux programmes : bg (background) et fg (foreground) La commande `ps aux` permet d'afficher l'etat d'execution des processus. Il affiche le status du processus :

— R : "runnable", veut dire que le processus est prêt à être éxécuté.
— S : "sleeping", veut dire que le processus est endormi.
— D : veut dire que le processus est sommeil interruptible.
— T : "traced", veut dire que le processus est arrété ou suivi.
— Z : zombie

Le + qui suit indique si le processus est en arrière plan.

4.3 Gestion des processus - Suite

4.3.1 Exercice 7 : Duplication de processus

L'objectif de cet exercice est de comprendre la notion de processus parent et de processus enfant lors de la duplication d'un processus par l'appel à la fonction `fork()`. Nous y revoyons la notion de PID et PPID (ou PID du parent) vus en cours. L'objectif est de mettre en évidence le role de l'appel à l'instruction fork() ainsi que la particularité de son code retour, et d'identifier que lors d'une duplication, le processus nouvellement créé ne reprend pas au début de son code mais poursuit l'exécution du processus parent. On rappelle que `fork()` retourne 0 dans le processus nouvellement créé (fils) et une valeur strictement supérieure à 0 sinon (qui correspond au PID du processus nouvellement créé)

4.3.2 Exercice 8 : Creation et destruction de processus

Cet exercice rappelle les notions de processus zombie et processus orphelin, et définit les modalités d'apparition de ces deux types particuliers de processus. En lançant la

commande `ps -aux` on peut observer le statut des processus.

1. Processus zombie

Un processus zombie est un processus présent alors que sont processus parent ayant terminé son exécution, il reste présent sur le système, en attente d'être pris en compte par son processus parent..

2. Processus orphelin

Un processus orphelin est un processus présent alors que sont processus parent est mort. Le processus orphelin est donc "adopté" par le processus 1, le processus "init"

4.3.3 Exercice 9 : Evaluation du nombre de processus

L'objectif de cet exercice est de comprendre le résultat de l'appel à l'instruction `fork()` ainsi que la particularité de son code retour, et d'identifier que lors d'une duplication, le processus nouvellement créé ne reprend pas au début de son code mais poursuit l'exécution du processus parent. On rappelle que `fork()` retourne 0 dans le processus nouvellement créé (fils) et une valeur strictement supérieure à 0 sinon (qui correspond au PID du processus nouvellement créé).

1. premier programme :

```
5 int main(){
6     fork();
7     fork();
8     fork();
9 }
```

Nous pouvons expliquer le code ci-dessus par le schema suivant :

FIGURE 4.1 – Duplication de processus 1

2. deuxième programme :

```
int main(){
    if(fork() > 0){
        fork();
    }
}
```

Nous pouvons expliquer le code ci-dessus par le schema suivant :

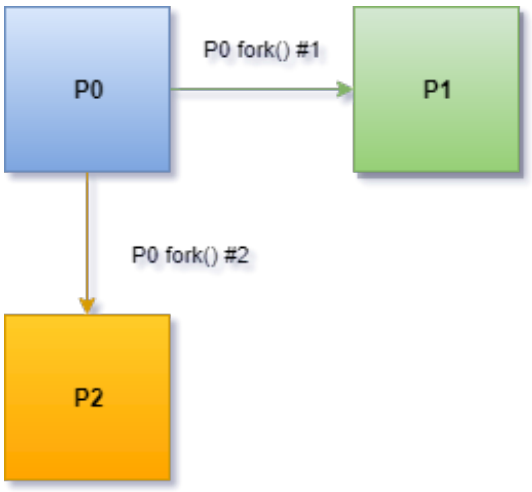

FIGURE 4.2 – Duplication de processus 2

3. troisieme programme :

```
int main(){
    int counter = 0;
    while(counter < 3){
        if(fork() > 0)
            counter++;
        else
            counter = 3;
    }
}
```

Nous pouvons expliquer le code ci-dessus par le schema suivant :

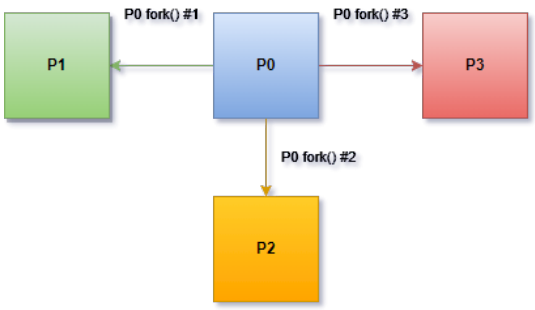

FIGURE 4.3 – Duplication de processus 3

4.3.4 Exercice 10 : Conjonctions, Disjonctions, et Duplication

L'objectif de cet exercice est de comprendre le résultat de l'appel à l'instruction `fork()` ainsi que la particularité de son code retour, et d'identifier que lors d'une duplication, le processus nouvellement créé ne reprend pas au début de son code mais poursuit l'exécution du processus parent.

En éxécutant quelques programmes exemple ci-dessous nous pouvons déterminer le fonctionnement des conjonctions et des disjonctions.

1. premier programme :

```
5 int main() {
6     fork() || (fork() && fork() );
7     exit(EXIT_SUCCESS);
8 }
```

1. Le processus père P0 : éxécute le premer `fork()` créant le processus P1 et retourne une valeur non-nulle et s'arrete.
2. Le processus fils P1 : la valeur du premier `fork()` est égale à 0 et évalue `&&`
3. Le processus fils P1 : éxécute donc le deuxième `fork()` fork créant P2 ainsi que le troisieme `fork()` créant P3
4. Le processus fils P2 : la valeur du deuxième `fork()` retourne 0, cour-circuite `&&` et s'arrete.
5. Le processus fils P3 : la valeur du troisième `fork()` retourne 0 et s'arrete.

Nous pouvons expliquer le code ci-dessus par le schema suivant :

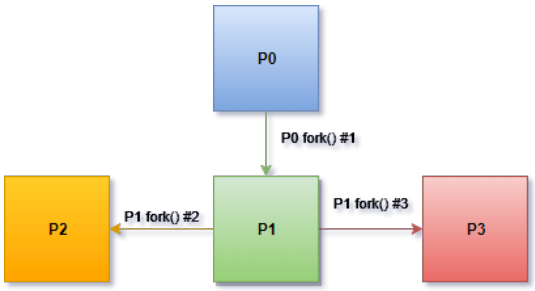

FIGURE 4.4 – Conjonction de processus 1

2. deuxième programme :

```
5 int main() {
6     fork() && (fork() || fork());
7     exit(EXIT_SUCCESS);
8 }
```

1. Le processus père P0 éxécute le premier `fork()` créant le processus P1 et retourne une valeur non-nulle.
2. Le processus fils P1 la valeur du premier `fork()` est égale à 0 et court-circuite `&&` et s'arrete.
3. Le processus père P0 : la valeur du premier `fork()` est non-nulle et évalue `||` et effectue le deuxième `fork`.

4. Le processus fils P2 : la valeur du deuxième `fork()` retourne 0 et exécute donc le troisieme `fork()` créant P3.
5. Le processus fils P2 s'arrete.
6. Le processus fils P3 s'arrete.

Nous pouvons expliquer le code ci-dessus par le schema suivant :

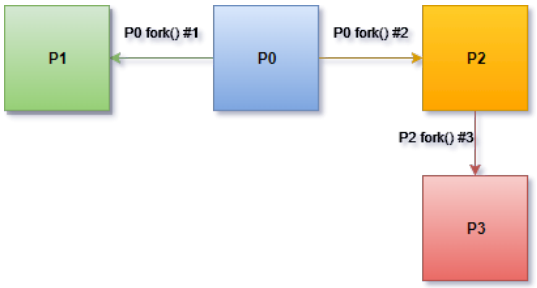

FIGURE 4.5 – Conjonction de processus 2

4.3.5 Exercice 11 : Terminaison normale de processus

L'objectif de cet exercice est de placer correctement les instructions `wait()`

Lorsque le processus fils se termine avant le processus père, il devient un zombie. Pour permettre à un processus fils en état zombie de disparaître complètement, on utilise la `wait()`.

1. premier programme :

```
5  int main() {
6      int result, a = 0;
7      result = fork();
8      if (result) wait(NULL);
9      if (result > 0)
10         a = 5;
11 }
```

2. deuxième programme :

```
5  int main() {
6      int result1, result2, result3;
7      result1 = fork();
8      result2 = fork();
9      result3 = fork();
10
11     if (result3) wait(NULL);
12     if (result2) wait(NULL);
13     if (result1) wait(NULL);
14 }
```

3. troisieme programme :

```
5  int main() {
6      int result1, result2, result3;
```

```
7       result1 = fork();
8       if (result1 == 0) {
9           result2 = fork();
10          if (result2 == 0) {
11              result3 = fork();
12              if (result3) {
13                  wait(NULL);
14              }
15              wait(NULL);
16          }
17          wait(NULL);
18      }
19  }
```

5. TP7+TP8 : Communication socket

5.1 Communication distante en utilisant l'outil netcat

5.1.1 Exercice 1 : Découverte de la commande nc : netcat

5.1.2 Exercice 2 : Utilisation de la commande nc : netcat pour le transfert de fichier et l'évaluation de la bande passante

5.1.3 Exercice 3 : Une histoire de serveurs concurrents ...

5.1.4 Exercice 4 : Comprendre une requête HTTP

5.2 Développement d'un client et d'un serveur en C

5.2.1 Exercice 5 : Mise en place d'une communication en mode non connecte

5.2.2 Exercice 6 : Création d'une architecture (client UDP) - (relai UDP-TCP)- (serveur TCP)

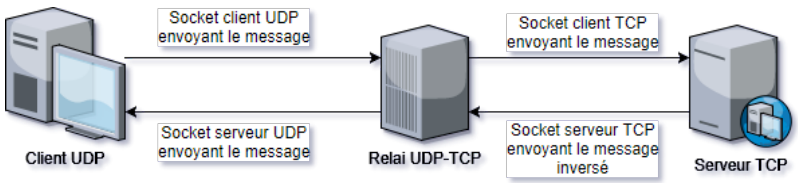

FIGURE 5.1 – Relai UDP-TCP

```
65      // Ecoute sur la socket.
66      listen(sp, 1);
67      //boucle infinie : nous écoutons les connexions entrantes sur la socket princip
68      while (1) {
69          // mode écoute : on attend l'arrivee d'une connexion entrante
70          if ((st = accept(sp, (struct sockaddr *) &cliAddr, &cliSize)) == -1) {
```

```
71              perror("erreur 2 accept()");
72              exit(1);
73          }
74
75          // réception d'un message
76          // message stocké dans buffer
77          // on lit alors sur cette socket un message
78          nbRecus = recv(st, buffer, MAXBUFFERSIZE, 0);
79          if (nbRecus == -1) {
80              perror("erreur recv()");
81              exit(1);
82          }
83
84          // on inverse le buffer :
85          inverser_buffer(buffer, strlen(buffer));
86          // ajout du caractère de fin de buffer :
87          buffer[nbRecus] = '\0';
88
89          //On renvoie le buffer inversé au client :
90          if (sendto(st, buffer, strlen(buffer), 0, (struct sockaddr *) &cliAddr, cli
91              perror("erreur sendto()");
92              exit(1);
93          }
94
95          // on ferme enfin la socket de travail
96          close(st);
97      }
98  }
```

5.3 Exercices bonus

5.3.1 Exercice 7 : Résolution de noms

Cet exercice à pour objectif de manipuler la fonction `gethostbyname()`. Cette fonction permet de transformer des noms de domaines en adresse ip, en interrogeant un serveur DNS.

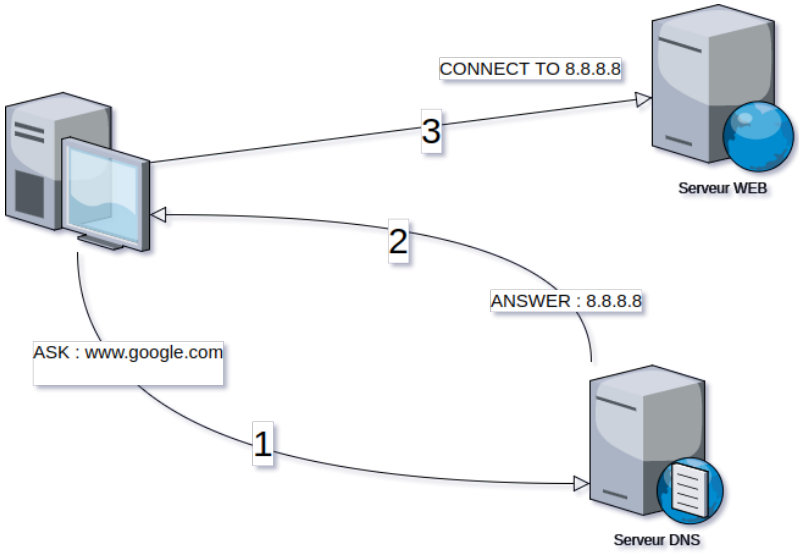

FIGURE 5.2 – Résolution DNS

A l'aide du manuel et des exemples disponibles sur internet, ainsi que de la documentation de la fonction `gethostbyname()` permettant la translation d'un nom de domaine vers une adresse IP. J'ai créé un programme qui affiche les adresses IP des noms de domaine "www.yahoo.fr", "www.gmail.com" et "www.u-bourgogne.fr".

```
10 void afficheIpHost(char *host) {
11     int i;
12     struct hostent *he;
13     struct in_addr **addr_list;
14     he = gethostbyname(host); // réupération des infos de l'host
15     if (!he) {
16         perror("Impossible de récupérer les infos host (vérifiez votre connexion in
17     } else {
18         // affichage de l'adresse IP :
19         printf("Adresse IP de %s : ", host);
20         addr_list = (struct in_addr **) he->h_addr_list;
21         for (i = 0; addr_list[i] != NULL; i++) {
22             printf("%s ", inet_ntoa(*addr_list[i]));
23         }
24         printf("\n");
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     char *yahoo = "www.yahoo.fr";
30     char *gmail = "www.gmail.com";
31     char *ubourgogne = "www.u-bourgogne.fr";
32
33     afficheIpHost(yahoo);
34     afficheIpHost(gmail);
```

```
35      afficheIpHost(ubourgogne);
36  }
```

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 :

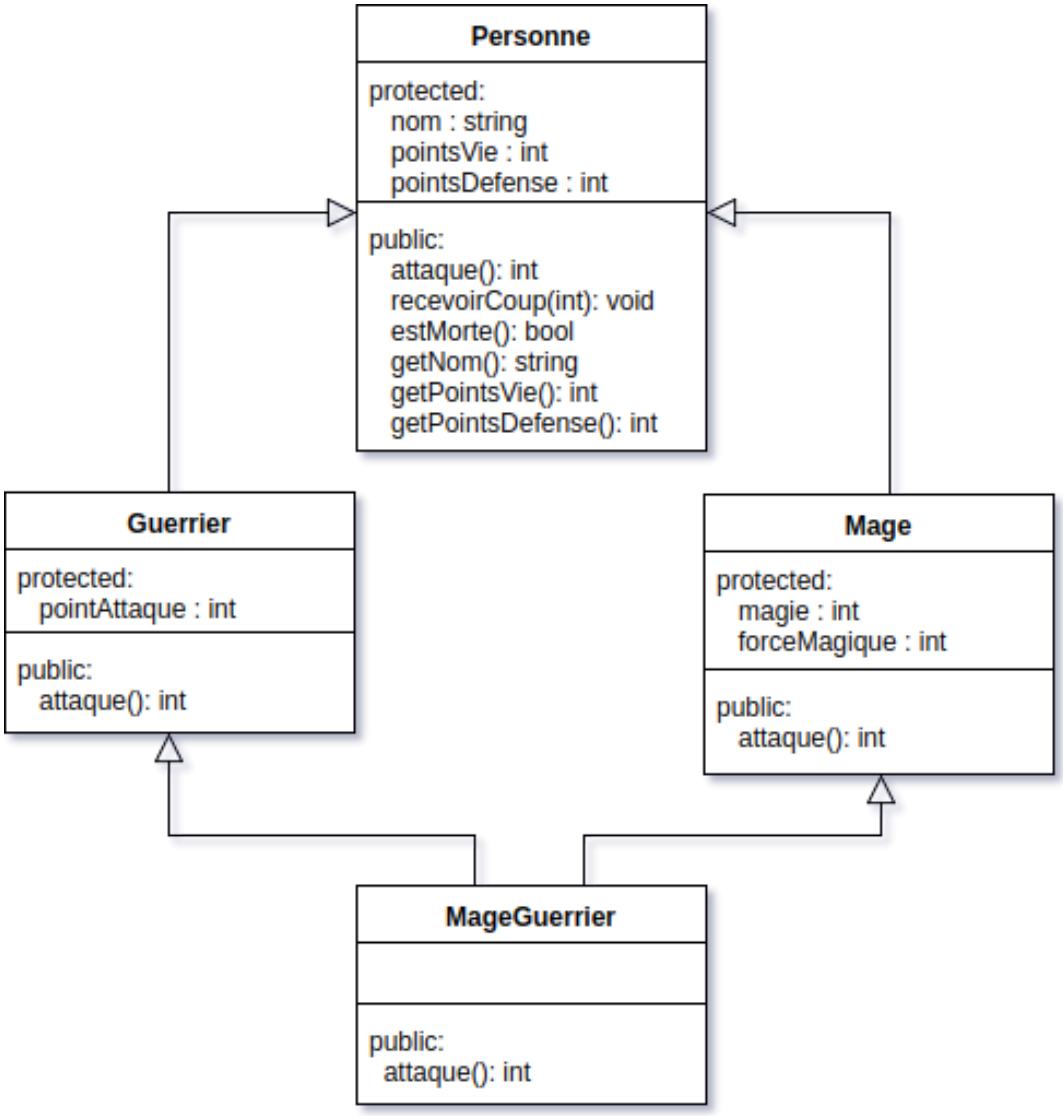

FIGURE 6.1 – Diagramme de classe

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;
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`.

```
23 }
24
25 bool Personne::estMorte() {
26     return (pointsVie <= 0);
27 }
```

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.

```
14 class Personne {
15 protected:
16     string nom;
17     int pointsVie;
18     int pointsDefense;
19
20 public:
21     Personne() = delete;
22
23     Personne(string nom, int pointsVie, int pointsDefense);
24
25     virtual ~Personne();
26
27     virtual int attaque()=0; //pure specifier
28
29     void recevoirCoup(int coup);
30
31     bool estMorte();
32
33     string getNom();
34
35     int getPointsVie();
36
37     int getPointsDefense();
38 };
```

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 :

```
12 class Guerrier : public virtual Personne {
13 protected:
14     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 :

```
MageGuerrier::MageGuerrier(string name, int pointsVie, int pointsDefense, int point
                           int forceMagique) :
        Personne(name, pointsVie, pointsDefense),
        Mage(name, pointsVie, pointsDefense, magie, forceMagique),
        Guerrier(name, pointsVie, pointsDefense, pointsAttaque) {
}

MageGuerrier::~MageGuerrier() {
    cout << "MageGuerrier is being deleted\n";
}

int MageGuerrier::attaque() {
    srand(time(NULL));
    return rand() % 2 ? Mage::attaque() : Guerrier::attaque();;
}
```

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.

```
int main() {
    displayHeader(1, (char *) "TP 9-10");
    bool finished = false;

    Guerrier *g1 = new Guerrier("Guerrier 1", 50, 5, 15);
    Mage *m1 = new Mage("Mage 1", 100, 3, 30, 12);
    MageGuerrier *mg = new MageGuerrier("MageGuerrier 1", 100, 4, 20, 10, 25);

    while (!finished) {
        combat(m1, mg);
        finished = endOfProgram();
    }

    delete g1;
    delete m1;
```

```
58      delete mg;
59
60      printf("Bye !!!");
61      exit(0);
62  }
```