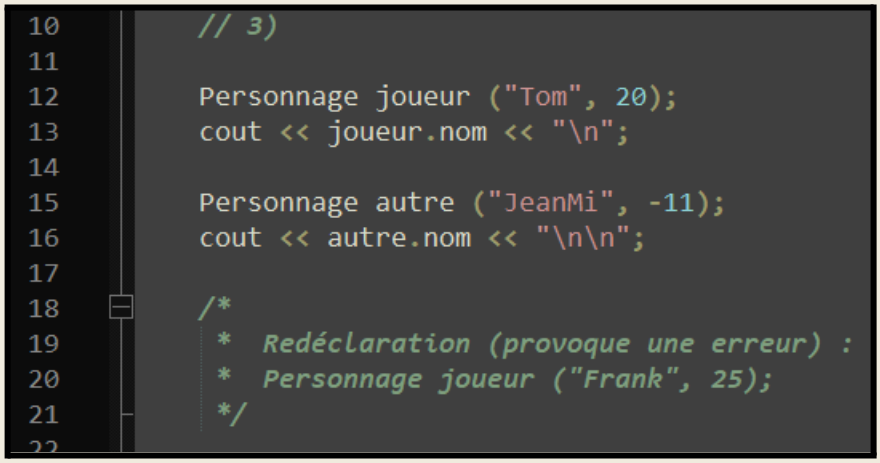
Sommaire

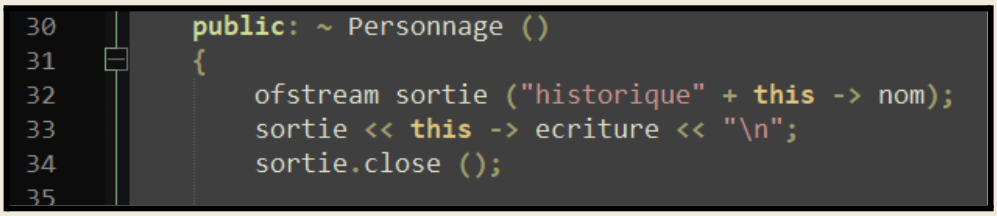
Constructeur -

En commençant par le constructeur de la classe pour la question 1.3, j'ai pu comprendre qu'il ne retournait rien et s'exécutait pendant l'instanciation de la classe, mais aussi qu'on ne pouvait pas appeler une 2e fois le constructeur : le compilateur ne réinstancie pas la classe, mais provoque simplement une erreur :



Destructeur -

De la même manière, le destructeur (écrit avec "~") s'exécute une seule fois juste avant de supprimer l'instance de la mémoire. Je m'en suis alors servi pour écrire l'historique des statistiques une et une seule fois dans un autre fichier ; en effet, il était impossible d'écrire dans une variable ofstream sans remplacer à chaque fois tout le contenu du fichier.



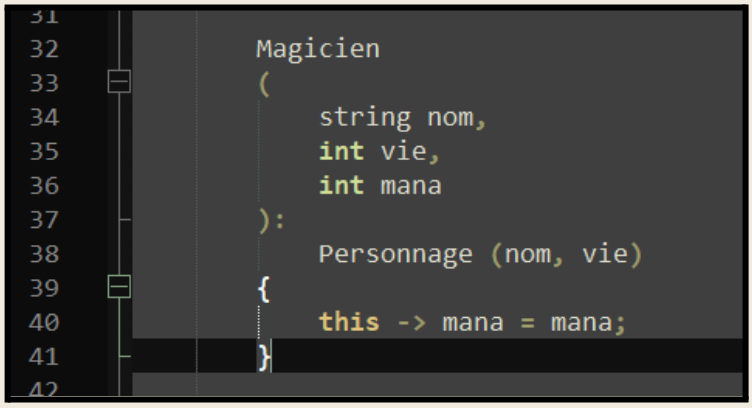
Définition des attributs -

J'ai rencontré des difficultés sur la manière d'accéder ou de redéfinir un attribut (".." pour un accès à un attribut d'une instance, "::" pour un accès statique, et "->" pour le pointeur "this" qui représente l'instance actuelle).

C'est pourquoi j'ai préféré définir tous mes attributs à l'intérieur de la classe, mais j'ai compris qu'il était aussi possible de la faire en dehors et donc de séparer en 2 fichiers déclaration et définition.

Héritage -

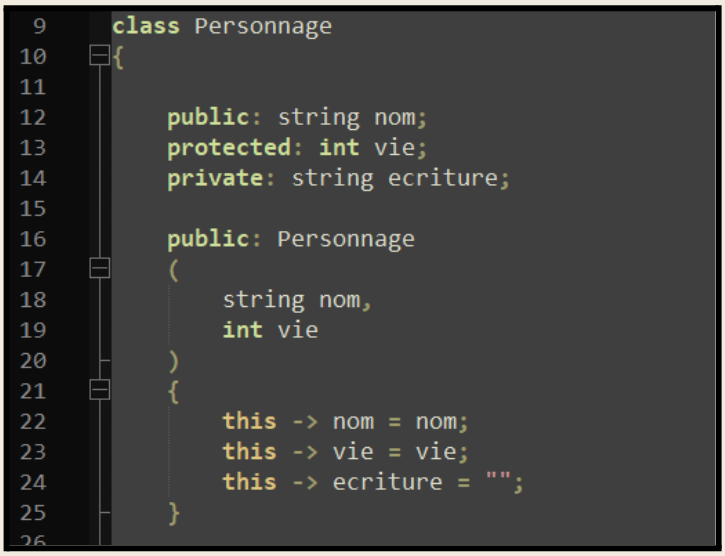
La classe LivretA hérite de la classe Banque grâce aux ":", de même que la classe Magicien hérite de Personnage ; tous ses attributs non privés sont alors utilisables par un Magicien (ce qui respecte la logique voulue par le TD), mais son constructeur est aussi spécifié de nouveau en se basant sur celui de Personnage, toujours avec les ":".



```
31
32          Magicien
33          (
34              string nom,
35              int vie,
36              int mana
37          ):
38              Personnage (nom, vie)
39          {
40              this -> mana = mana;
41          }
42
```

Encapsulation -

Pour un 1er essai avec la classe Personnage, j'ai commencé par définir la privacité de chaque attribut à chaque fois pour être plus explicite, mais cela s'est révélé être fouilli au fur et à mesure que les attributs s'ajoutaient :



```
9   class Personnage
10  {
11
12      public: string nom;
13      protected: int vie;
14      private: string ecriture;
15
16      public: Personnage
17      (
18          string nom,
19          int vie
20      )
21      {
22          this -> nom = nom;
23          this -> vie = vie;
24          this -> ecriture = "";
25      }
26
```

En regardant d'autres classes sur Internet, j'ai alors compris qu'il était plus simple de séparer la déclaration en 3 parties (public, protected et private) ; bien-sûr, tout accès à un attribut privé en dehors de la classe provoque une erreur, comme dans le test de la question 1.6 :

```
class Banque
{

    protected:

        double solde;


    public:

        Banque ()
        {
            this -> solde = 1000.;
        }


        // 3.1)

        void deposerArgent
        (double montant)
        {
            this -> solde += montant;
        }
```

```
    // 6)

    /*
     *  Réattribution privée (provoque une erreur) :
     *  autre.vie = 17;
     */
```

Ensuite, j'ai aussi pu comprendre l'intérêt des attributs en protected (qui sont comme les privés, mais accessibles par les classes qui en héritent) ; ils peuvent s'avérer utiles, entre autres choses, pour une variable sur les points de vie, qui sont évidemment inaccessibles par l'utilisateur de la classe, mais peuvent être altérés par un pouvoir de magicien par exemple.

Opérateurs -

Pour la question 1.9, je n'ai pas trouvé comment redéfinir un opérateur ("<<") déjà redéfini dans la même classe : le lancement produisait à chaque fois une erreur disant qu'il manquait le paramètre joueur (décrit comme argument pour l'ancien opérateur "<<"), mais nous voulons justement un opérateur sans paramètre supplémentaire. Je l'ai alors renommé en "&".

```
    // 1.9)

    public: void operator & ()
    {
        ecriture += this -> stat ();
    }
```

Fonctions virtuelles -

La fonction changeVie doit pouvoir être surchargée et définie différemment dans la classe Magicien ; comme vu dans les questions 4.9 et 4.10, ceci n'est possible qu'en déclarant la fonction mère comme "virtual" dans la classe Personnage. Ainsi la fonction fille passera par dessus, mais la fonction mère sera quand même appelée pour un Personnage non Magicien, ou simplement si la fonction fille n'a pas été définie.

```
85      public: void virtual changeVie
86      (int nouvVie)
87      {
88
89          // 2.3)
90
91          if (nouvVie < 0)
92          {
93              wcout << "Attention : la vie d'un personnage ne peut etre negative.\n";
94              this -> vie = 0;
95          }
96          else
97          {
98              this -> vie = nouvVie;
99          }
100     }
```

```
Attention : la vie d'un personnage ne peut etre negative.
Golem -
Points de vie : 0

Jclaude -
Points de vie : 3
Points de mana : 10

Golem -
Points de vie : 26

Attention : la vie d'un magicien ne peut etre negative.
Jclaude -
Points de vie : 0
Points de mana : 10
```

Mais deux fonctions de même nom seront tout de même considérées comme à part si elles ne prennent pas les mêmes arguments, ce qui paraît logique.

Attributs statiques -

Pour la question 4.4, j'ai aussi eu besoin d'une fonction trimJump qui enlève un saut de ligne à la fin d'une chaîne pour reformater les statistiques de la classe Personnage, et ainsi pouvoir les réutiliser dans la méthode stat du Magicien :

```
103     // 4.4)
104
105     public: static string trimJump
106     (string chainJump)
107     {
108         uint newLength = chainJump.length () - 1;
109
110         if (chainJump [newLength] != '\n')
111         {
112             cout << "Retranchement d'un caractère différent de '\\n'";
113             exit (0);
114         }
115
116         return chainJump.substr (0, newLength);
117     }
```

```
49          // 4.4)
50
51          string stat ()
52          {
53              return
54                  Personnage:: trimJump (Personnage:: stat ())
55                  + "Points de mana : "
56                  + to_string (this -> mana)
57                  + "\n\n"
58              ;
59          }
```

La déclaration statique peut être pratique pour ce genre de fonctions génériques ne dépendant d'aucune instance.

Aléatoire -

Pour les questions 4.7 à 4.9, j'ai utilisé la manière classique de générer un nombre aléatoire avec "rand ()" qui retourne un entier aléatoire ; j'ai alors eu besoin de moduler par 101 pour restreindre le résultat entre 0 et 100, avant de le translater par -50.

```
37      // 7 et 8)
38
39      srand (time (NULL));
40      int randomNb;
41
42      for
43      (int _ = 0; _ < 5; _ ++)
44      {
45          randomNb = rand () % 101 - 50;
46          sorcier1.changeVie (randomNb);
47          cout << sorcier1.stat ();
48
49          randomNb = rand () % 101 - 50;
50          sorcier2.changeVie (randomNb);
51          cout << sorcier2.stat ();
52      }
```