NOM : Prénom :

Examen Méthodologie Orientée Objets (IF3)

Durée 1h, documents autorisés, annales et appareils communiquant interdits

→Répondre sur une nouvelle copie double et rendre le sujet←

Exercice 1 : Trouver les erreurs (15 minutes)

1) Les programmes suivants possèdent chacun ***une*** erreur de conception (*i.e.* une erreur lors de la compilation, de l'exécution, résultat incohérent). Pour *chaque* programme, **indiquer** quelle est l'erreur et **corriger** la (directement sur l'énoncé).

```
#include <iostream>
using namespace std;
int main()
{ double tab[]={1,2,3,4,5};
  cout << "valeurs : " << endl;
  for(int i=0; i<6; i++ )
     cout << tab[i] << " ";
  return 0;
}
```

Programme 1

```
double * AllocDyn(int t)
{       double *tab = new double[t];
        return tab ;              }
int main()
{       double *t=NULL;
        t = AllocDyn( 5 );
        t[0] = 0;
        return 0;
}
```

Programme 2

```
#include <iostream>
using namespace std;
int main()
{
        double *a = 3.14;
        cout << *a << endl;
        return 0;
} // affiche n'importe quoi et
 //  plante souvent
```

Programme 3

```
#include <iostream>
using namespace std;
unsigned int ABS(int a, int b)
{   unsigned int r = a-b;
    return r;                      }
int main()
{ int u=3, v=5;
  cout<<"|u-v|="<< ABS(u,v) <<endl;
  return 0;
}  // affiche: |u-v|=4294967294
```

Programme 4

2) **Modifier les classes B** suivantes pour que les programmes compilent, s'exécutent et produisent un résultat cohérent. Il n'y a qu'**une** erreur par classe B.

```
#include <iostream>
using namespace std;

class B
{
double Xb;
  public:
     B():Xb(1) {}
     B(B b) {Xb = b.Xb;}
};

int main()
{
  B b1;
  B b2(b1);
  return 0;
}
//Erreur de compilation ligne :
//      B(B b) {Xb = b.Xb;}
```

Programme 5

```
#include <iostream>
using namespace std;

class A {
  public:
    double Xa;
    A(const double &x):Xa(x) {}
};

class B : public A
{ public :
    double Xb;
    B(const double &xb=0):Xb(xb) { }
 };

int main()
{   B b;
    cout << b.Xa << b.Xb << endl;
    return 0;
}
// Erreur de compilation ligne : B b ;
```

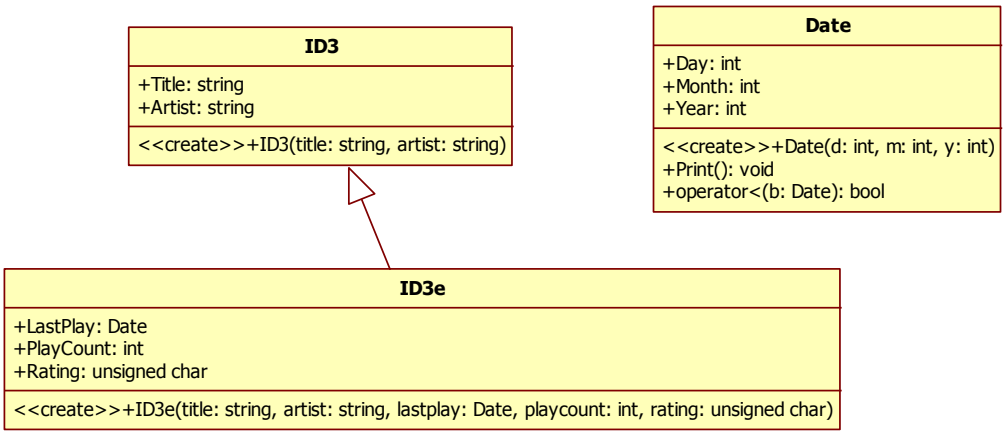
Programme 6

Exercice : Smart PlayList MP3 (45 minutes)

Une start-up développe un nouveau lecteur MP3 qui s'adapte aux préférences de l'utilisateur. Classiquement un titre musical contient des métadonnées (comme le titre, artiste, album, image, année, genre…) définies dans le standard ID3. La start-up veut ajouter des informations aux métadonnées ID3 afin de créer des playlists basées sur les préférences de l'utilisateur.

Classes ID3, ID3e et Date (10 minutes)

Chaque MP3 contient un ID3. Les métadonnées présentent dans un ***ID3*** sont très nombreuses. On ne considère ici que les champs *Title* et *Artist* comme métadonnées.
Les préférences de l'utilisateur sont modélisées par la classe ***ID3e*** qui hérite de ***ID3*** et ajoute les informations de date de la dernière lecture (*LastPlay*) du morceau, du nombre d'écoutes (*NbPlay*) du morceau et de l'avis de l'utilisateur (*Rating*) sur le morceau. C'est grâce à ces informations que la playlist basée sur les préférences de l'utilisateur sera construite.



Les classes ID3 et ID3e n'ont pas de fonction membre hormis leur constructeur. Les fonctions permettant d'utiliser ces classes (affichage, accesseurs), spécifiques au système et non aux données ID3, seront réalisées dans les classes dites « API » : ApiID3 et ApiID3e (partie suivante).

1) Donner le code C++ de la définition de la classe ***ID3e***. Préciser le code du constructeur.
2) Quelle relation existe entre ***ID3e*** et ***Date*** ? Justifier et modifier le diagramme UML pour faire apparaitre ce lien (directement sur l'énoncé).
3) Donner le code C++ de l'opérateur < de la classe ***Date***. Il retournera *true* que si la date de l'objet appelant est plus ancienne que la date de l'objet passé en paramètre.

Classes ID3, ID3e et Date (15 minutes)

La classe C++ ***ApiID3*** qui ajoute les fonctionnalités à ***ID3*** est donnée ci-après:

```
class ApiID3
{ protected:
    ID3 * Tag;
  public:
    const string & GetTitle() const  { return Tag->Title;  }
    const string & GetArtist() const { return Tag->Artist; }
    virtual const ID3* GetInfo() const { return Tag;}
    void SetID3(ID3 *t) { Tag = t;}
    ApiID3(ID3 *t) {Tag = t;}
    virtual void Print() { cout << GetTitle() <<", "<<GetArtist()<< endl; }
};
```

4) Donner le diagramme UML de cette classe. Faire apparaitre par la relation le champ *Tag* et justifier rapidement la relation utilisée.
5) A quoi serve chacun des mots clés `const` présents dans la déclaration de la fonction *GetInfo()* ?
6) La classe ***ApiID3e*** hérite de ***ApiID3***. Il faudra surcharger les méthodes virtuelles, et pour *GetInfo()* retourner un pointeur de type ***ID3e***.
   a. Modéliser (UML) la classe ***ApiID3e***.
   b. Donner le code C++ du constructeur.
7) Etant donné le code ApiID3e::*Print*() ci-après :

```
virtual void ApiID3e::Print()
{ ApiID3::Print();
  cout << "Last Play on : "; GetInfo()->LastPlay.Print(); // affichage: JJ/MM/AAAA
  cout << ", Played : "; cout << GetInfo()->PlayCount ; cout << "times";
  cout << ", rating = "; cout << (int) GetInfo()->Rating << endl ; }
```

Qu'affiche le programme suivant ? On rappelle que lors d'une surcharge de fonction, le type de retour n'est pas pris en compte.

```
ID3  m1("Dimitri", "Barbarian Peace" );
ID3e m3("Deletere", "Les Olibrius", Date(21,6,2016), 12, 200);
ApiID3 api1(&m1);
Api1.Print();
ApiID3e apie3(&m3);
Apie3.Print();
```

Classe SmartPlayList (15 minutes)

La classe ***SmartPlayList*** génère une playlist basée sur les préférences de l'utilisateur.
Cette classe récupère des objets ***ID3e***. Les champs *LastPlay*, *Rating*, *PlayCount* sont mis à jours par d'autres constituants du système.
Cette classe aura comme fonctionnalités :

- Ajoute un morceau à la playlists (via un objet ***ID3e***) : *AddTrack()*
- Afficher les morceaux : *Print()*
- Trier les morceaux en modifiant l'ordre de la playlist : *OrderPlayList()*

Voici son code C++, ainsi qu'une partie de la fonction *Print()* :

```
class SmartPlayList
{    vector< ApiID3e *> Tracks;
   public:
    void AddTrack( ApiID3e * t) { Tracks.push_back( t ); }
    void OrderPlayList();
    void Print() const;   };
void SmartPlayList::Print() const
{   cout << "Tracks order:";
    for (unsigned int i=0; i<Tracks.size(); i++)
                Print(); // <- completer cette ligne
    cout << "\n";
}
```

8) Compéter la ligne de la fonction *SmartPlayList ::Print()* afin que tous les éléments de *Tracks* soient affichés.
9) `OrderPlayList()` va modifier le tableau *Tracks* en ordonnant les éléments suivant l'ordre chronologique (on simplifie ici la formule réelle s'appuyant aussi sur *Rating* et *PlayCount*). A partir de la documentation de ***sort*** fournie ci-après, écrire la fonction *SmartPlayList ::OrderPlayList()* et la/les fonctions permettant son fonctionnement.

std::sort

<algorithm>

```
default (1)   template <class RandomAccessIterator>
                void sort (RandomAccessIterator first, RandomAccessIterator last);
custom (2)    template <class RandomAccessIterator, class Compare>
                void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Sort elements in range

Sorts the elements in the range `[first,last)` into ascending order.

The elements are compared using `operator<` for the first version, and *comp* for the second.

Equivalent elements are not guaranteed to keep their original relative order (see `stable_sort`).

Parameters

`first, last`

Random-access iterators to the initial and final positions of the sequence to be sorted. The range used is `[first,last)`, which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.
`RandomAccessIterator` shall point to a type for which `swap` is properly defined and which is both *move-constructible* and *move-assignable*.

`comp`

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.
The function shall not modify any of its arguments.
This can either be a function pointer or a function object.

Example

```
1  // sort algorithm example
2  #include <iostream>     // std::cout
3  #include <algorithm>    // std::sort
4  #include <vector>       // std::vector
5
6  bool myfunction (int i,int j) { return (i<j); }
7
8  struct myclass {
9    bool operator() (int i,int j) { return (i<j);}
10 } myobject;
11
12 int main () {
13   int myints[] = {32,71,12,45,26,80,53,33};
14   std::vector<int> myvector (myints, myints+8);               // 32 71 12 45 26 80 53 33
15
16   // using default comparison (operator <):
17   std::sort (myvector.begin(), myvector.begin()+4);           //(12 32 45 71)26 80 53 33
18
19   // using function as comp
20   std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)
21
22   // using object as comp
23   std::sort (myvector.begin(), myvector.end(), myobject);     //(12 26 32 33 45 53 71 80)
24
25   // print out content:
26   std::cout << "myvector contains:";
27   for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
28     std::cout << ' ' << *it;
29   std::cout << '\n';
30
31   return 0;
32 }
```

Output:

```
myvector contains: 12 26 32 33 45 53 71 80
```

Fin.