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.

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

Programme 1

```
void AllocDyn(double *tab, int t)
{       tab = new double[t]; }

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

Programme 2

```
int main()
{
        double a = 3.14;
        double *n = &a;
        double r = 1.0/*n;
        return 0;
} // ne compile pas !
// d'après compilateur, il y a une
// erreur ligne 5 « double r =… »
```

Programme 3

```
#include <iostream>
using namespace std;
double MyFunc(int a, int b)
{return 1.0*a/b;}

int main()
{  cout << MyFunc(1.0,3.1415);
   return 0;
}// affiche 0  ☹
```

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 X;
        double GetX2()
           { return X*X; }
};

int main()
{
        B b;
        b.X = 2;
        cout << b.GetX2();
        return 0;
}
//Erreurs de compilation lignes :
//   b.X = 2;
//   cout << b.GetX2();
```

Programme 5

```
#include <iostream>
using namespace std;
class B
{   double X;
public:
   void SetX(double x) { X=x; }
   double GetX()    { return X; }
};
class C : public B
{public:
        void Affiche() const
         { cout << GetX(); }
};
int main()
{       C c;
        c.SetX(3);
        c.Affiche();
        return 0;
} // Erreur de compilation ligne
//   { cout << GetX(); }
```

Programme 6

Exercice : Diagramme d'états (45 minutes)

Le but de cet exercice est de concevoir et programmer des classes permettant de modéliser des diagrammes d'états puis de les exécuter.

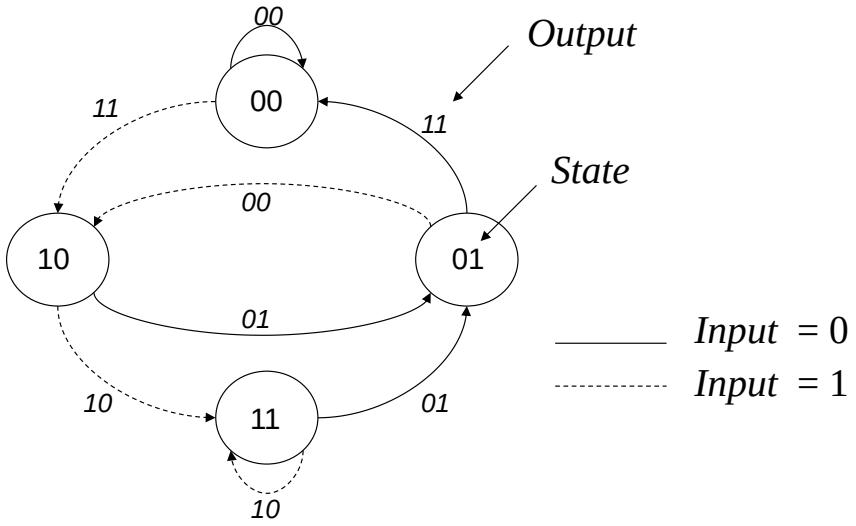
Une machine d'états (ou automate fini) est constituée d'un ensemble d'états et d'un ensemble de transitions. Les transitions relient un état d'origine vers un état d'arrivé.

Un diagramme d'états représente les états (***State***), les transitions (***Transition***) et les comportements d'une machine d'états.

Une machine d'états passe d'un état à un autre, via une transition, en fonction des valeurs d'entrées (***Input***). Lors d'une transition, des valeurs de sortie (***Output***) sont produites.

Dans l'exemple suivant, un diagramme d'états à 4 états est représenté, avec ses transitions et les deux bits produits lors d'une transition. Dans cet exemple, l'entrée n'est que d'un bit.

Ainsi, si on se trouve à l'état « 01 » et qu'un 0 arrive en entrée, alors on passe à l'état « 00 » en produisant en sortie « 11 ».



Classes SDObjects, Transition et State (20 minutes)

Voici les entêtes des classes fondamentales utilisées dans la modélisation des diagrammes.

```
class SDObject
{ public:
    string Name;
    vector<bool> Value;
    void PrintVectorBool() const;
    virtual void Print() const = 0;
    SDObject(const string &n, const
vector<bool> &r);
    virtual ~SDObject();
};
```

```
class Transition: public SDObject
{private:
    State *GoToState;
public:
    Transition(const vector<bool> &v, const
State* from, State *to);
    State * GetGoToState() {return
GoToState;}
    void Print() const;
};
```

```
class State: public SDObject
{ private:
    vector<Transition*> GoByTransition;
  public:
    State(const string &n, const vector<bool> &r): SDObject(n,r) {}
    virtual ~State();
     Transition * GetTransitionForValue(unsigned int in) {return GoByTransition[in];}
     void AddSelfLoop  ( const vector<bool> &v );
     void AddTransition(const vector<bool> &v, State *toState);
     void Print() const;
};
```

On donne aussi le code du constructeur de ***Transition*** :

```
Transition::Transition(const vector<bool> &v, const State* from, State *to):
SDObject(from->Name + "->" + to->Name, v)
{   GoToState = to; }
```

et le code la fonction *AddTransition* de la classe **State** qui permet l'ajout d'une transition à un état :

```
void State::AddTransition(const vector<bool> &v, State *toState)
{
    Transition *t = new Transition(v, this, toState);
    GoByTransition.push_back(t);
}
```

A noter :

- le champ *Name* permet de donner un nom à une transition ou un état.
- Le champ *Value* permet de coder une valeur binaire sous forme d'un tableau de booléens. Celle-ci correspondra soit à la valeur d'un état, soit à la valeur à produire en sortie pour une transition.

1) Donner la modélisation UML des trois classes ***SDObject***, ***State*** et ***Transition***.
   a. Préciser les spécificités de ***SDObject***
   b. Expliciter et justifier les relations entre les classes ***State*** et ***Transition.***
2) Justifier la présence d'un destructeur dans la classe ***State***.
3) Ecrire le code C++ de la fonction de *State::AddSelfLoop* qui permet d'ajouter une transition (et une valeur) à un état sur lui-même (sur l'exemple : états « 00 » et « 11 »).
4) Compléter le code suivant (~8 lignes) afin de reproduire le diagramme donné en exemple. Attention à l'ordre des transitions…

```
vector<State> S;
S.push_back( State ("A",{0,0}) );
S.push_back( State ("B",{1,0}) );
S.push_back( State ("C",{1,1}) );
S.push_back( State ("D",{0,1}) );
```

Classe StateDiagram (20 minutes)

On souhaite maintenant ajouter une classe ***StateDiagram*** pour le concept de diagramme d'états. Cette classe permettra de contenir un diagramme d'états et de le faire fonctionner par rapport à une entrée. Notamment, il faudra pouvoir :

- se souvenir de l'état courant (celui qui est actif)
- initialiser à l'état « 00 » le diagramme
- disposer de la méthode ***RunOnce*** qui permet de changer d'état à partir d'une valeur d'entrée (*input*) sous forme d'un tableau de booléens ou d'un entier. Cette fonction retourne la valeur de la transition.
- disposer de la fonction ***PrintRun*** qui à partir d'un tableau de *n* valeurs d'entrée, affiche successivement les *n* valeurs des transitions effectuées.

5) **Donner un diagramme de classes UML de la classe *StateDiagram*** s'appuyant sur les trois classes précédentes. Justifier les relations entre ***StateDiagram*** et les autres classes.
6) Ecrire la fonction *StateDiagram ::RunOnce(int in) ;*
7) Ecrire la fonction *StateDiagram ::PrintRun*

Fin.