Objets Simples & Opérateurs

Pour valider la description d'un circuit intégré, il est nécessaire de passer par une étape de simulation (cf. `asimut`). Une variable à l'intérieur du simulateur est un booléen, mais dans certains cas on peut ne pas connaître sa valeur. Le simulateur introduit donc un nouveau type de variable pouvant prendre trois valeurs: **0**, **1** ou **U** (`Undefined` ou encore `non-définie`).

Au cours de ce TME nous allons créer ce nouveau type (ou classe) que nous appellerons `LogicValue`.

Spécification de `LogicValue`

La classe `LogicValue` va contenir un seul attribut `_value` qui sera de type entier et ne pourra prendre que trois valeurs: 0, 1 ou 2 ayant respectivement la signification `Zero`, `One` et `Undefined`.

Plutôt que travailler avec des constantes numériques, nous allons utiliser des constantes nommées. On réalise cela avec un `enum`, comme ci-après:

```
class LogicValue {
  public:
    enum Value { Zero=0, One=1, Undefined=2 };
  public:
    // ...
};
```

A l'extérieur de la classe on les référencera avec la syntaxe `LogicValue::Undefined` et à l'intérieur, simplement comme `Undefined`.

La classe `LogicValue` possèdera les fonctions membres suivantes:

Constructeurs: (*CTOR*)

- `LogicValue()`, le constructeur par défaut, devra affecter la valeur `Zero`.
- `LogicValue{{{int`, un contructeur à partir d'un entier ordinaire. Tout entier de valeur supérieure ou égale à 2 sera considéré comme `Undefined`.
- `LogicValue(const LogicValue&)`, un constructeur par copie.

Destructeur: (*DTOR*)

- `~LogicValue()`, le destructeur (unique).

Accesseurs: (*Accessors*)

- `print(std::ostream&)`, une fonction d'affichage.
- `toInt()`, une fonction de conversion vers un entier.

Modifieur: (*Mutators*)

- `fromInt(int)`, affectation depuis un entier.

En plus des fonctions membres listées çi dessus, on ajoute les fonctions non-membres suivantes permettant de réaliser des opérations logiques:

- `LogicValue non(const LogicValue&)`, négation logique.
- `LogicValue ou (const LogicValue&, const LogicValue&)`, *ou* logique.
- `LogicValue et (const LogicValue&, const LogicValue&)`, *et* logique.

La table de vérité des fonctions négation et *OU* vous est fournie çi-après.

Question 1

Implanter la class `LogicValue` telle que décrite précédemment. Comme au TME1, on tracera les appels aux constructeurs et destructeurs. Tester à l'aide du petit programme de test suivant:

Retrouvez la table de vérité classique (sans états non-définis) avec la commande **shell** suivante:

```
./boolvalue | grep -v "U"
```

Question 2

Écrire la table de vérité de la fonction *ET* puis l'implanter. Vérifier à l'aide d'un programme de test calqué sur celui du *OU* que la fonction se comporte correctement.

Question 3

En vous basant sur l'opérateur qui a été présenté en cours, écrire l'opérateur d'écriture dans un flot pour l'objet `LogicValue`. Modifier le code des fonctions de vérification en conséquence.

Question 4

Écrire l'opérateur d'affectation, modifier le code de test en conséquence. A quelle fonction membre se substitue-t-elle?

Question 5

Écrire les fonction (non-membre) *`ou`* et *`et`* sous forme d'opérateurs:

```
LogicValue operator or  ( const LogicValue&, const LogicValue& );
LogicValue operator and ( const LogicValue&, const LogicValue& );
```

Modifier le code de test.