Université de Paris

Rapport de Projet

Sujet n°2
-
Simulateur de circuits combinatoires

SCHMIT Madeleine, DIDIER Quentin
M1 - MIC

Sommaire

1. Introduction

Dans le cadre du cours Langages à Objet Avancés, nous avons consacré plusieurs semaines de travail au développement d'un simulateur de circuits combinatoires en utilisant les diverses connaissances que nous avions pu nous approprier tout au long du premier semestre. Les concepts au cœur de notre projet ont principalement été la hiérarchisation de diverses classes représentant les différents types de portes logiques, une modélisation en structure d'arbre qui se rapproche fortement du design pattern 'Factory Method' et à laquelle nous nous sommes d'autant plus familiarisés au cours des dernières semaines.

Tout au long du projet, des choix d'implémentation ont dû être faits en accord avec la vision de chacun, donnant naissance à des argumentations profondes nous poussant à analyser le sujet et les diverses implémentations possibles en détail. Le travail en binôme nous a donc souvent permis d'échanger de nouvelles perspectives concernant le développement du projet et d'aiguiller nos connaissances respectives.

Ces discussions sont nées de divers questionnements face à un sujet qui nous semblait à première vue libre d'ambiguïté, mais qui malgré tout a fait surgir différentes formes d'interprétations possibles: Quelles sous-classes regroupent un circuit et comment représenter ce dernier ? Quelle différence de design et de hiérarchisation doit-on adopter pour les différentes catégories de portes logiques ? Il s'agit ici de diverses questions très importantes dans la réalisation d'un projet performant et répondant aux différents besoins formulés dans l'énoncé du sujet. Il est donc primordial d'aborder ces interrogations avec une réflexion profonde et d'adopter une organisation supervisée du travail à réaliser.

Qui dit projet informatique, dit donc gestion de projet. Tout projet aboutissant à un résultat satisfaisant les deux parties d'un binôme, ainsi que le client auquel est destiné le livrable, est nécessairement le fruit d'une entente basée sur une communication saine et une écoute active de son coéquipier, ainsi que sur une collaboration organisée.

La réalisation de ce projet basé sur la conception d'un circuit combinatoire a donc fait surgir divers questionnements, ainsi que des aspects fondamentaux dans la gestion de ce dernier, avant d'aboutir à un résultat final satisfaisant nos attentes mutuelles, des points clés qui seront détaillés dans la suite.

2. Choix du sujet

(*Réflexions et questionnement sur notre choix de ce sujet et nos objectifs initiaux*)

Notre choix concernant le sujet du projet, s'est immédiatement porté sur la réalisation d'un simulateur de circuits combinatoires, après une lecture et analyse profonde des différentes thématiques proposées. Opter pour la programmation d'un tel circuit booléen nous paraissait en effet le plus logique, puisque c'était ce sujet en particulier qui éveillait en nous le plus grand intérêt, compte tenu des cours que nous avons suivis ce semestre, comme par exemple le cours de *Complexité*, qui s'appuyait à de nombreuses reprises sur de tels circuits pour justifier les taux de complexité obtenus.
Par ailleurs, la conception d'un programme basé sur une structure arborescente, faisant ainsi intervenir la hiérarchisation des classes ainsi que le concept des classes abstraites, nous semblait réellement pertinente, dans l'optique de pouvoir mettre en œuvre les diverses connaissances cumulées lors de ce semestre en Langages orientés Objets.
Finalement, la liberté qui nous a été laissée dans la modélisation du sujet et dans la conception des différentes classes nous a permis de nous rapprocher d'une problématique réelle à laquelle nous pourrions faire face dans un futur proche : partir d'une demande d'un client sans instructions détaillées, qui est plutôt simple et omise de consignes ayant pour but de nous guider pas par pas dans nos choix, afin d'aboutir de nous même à un produit final, en nous basant sur nos propres choix d'implémentation et de modélisation UML.
Le principal est donc d'avoir une vision claire du résultat que l'on souhaite obtenir à partir d'une demande formulée de manière simple, chose qui nous a été permise par le sujet en question, mais surtout de se donner les moyens de réaliser cette dernière, afin de livrer avant l'échéance un produit fonctionnel et performant.

3. Gestion de projet

(*Explications des méthodes et outils utilisés pour organiser notre travail en collaboration*)

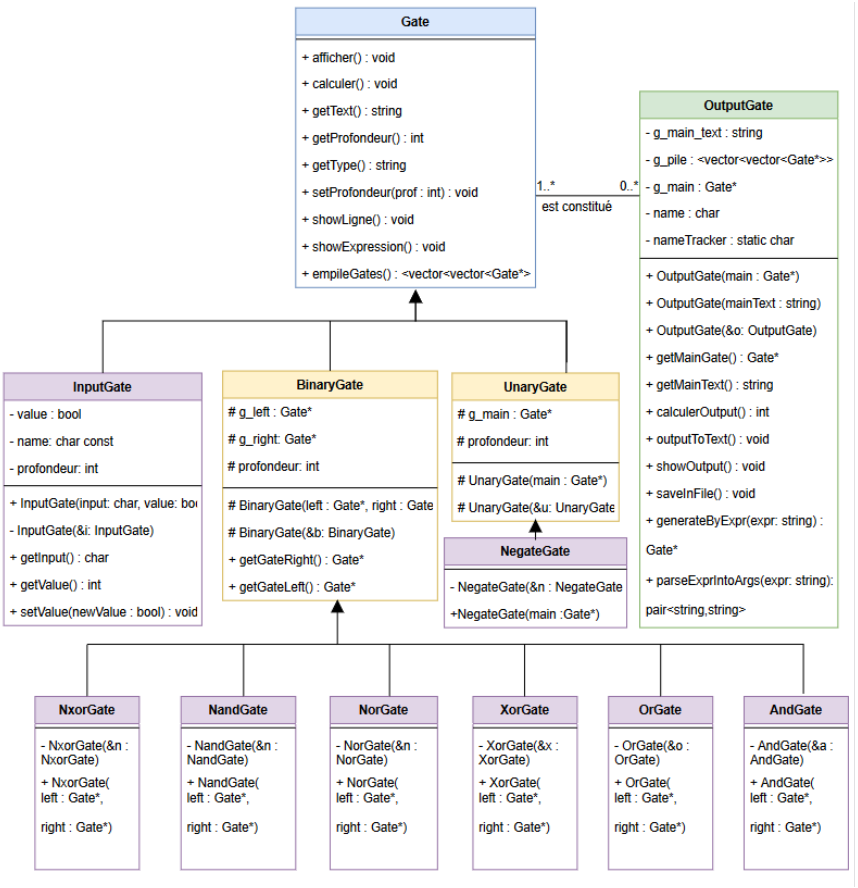
Afin d'atteindre notre vision d'un projet répondant aux différentes attentes fonctionnelles présentées dans le sujet, il a été primordial de se consacrer dans un premier temps à l'appropriation de bonnes pratiques concernant la gestion de projet.
Devant travailler majoritairement à distance à cause de la pandémie et de nos pays de résidence différents (Luxembourg - France), le travail collaboratif n'a pas été des plus faciles. Cependant il a fallu s'y adapter rapidement en optant pour le système de gestion de version Git, ainsi que la plateforme Github, ce qui nous a permis de contourner certaines difficultés liées à un travail de groupe à distance.
Par ailleurs, une première réunion audiovisuelle nous a permis de nous mettre d'accord sur la façon de vouloir aborder le sujet, c'est-à-dire la compréhension des différentes notions, la structure UML de notre futur projet, la mise au clair de désaccords concernant cette dernière, ainsi que la répartition des premières tâches à réaliser dans un délai de 3 jours.
Après ce premier *Micro-Sprint*[1], qui nous a permis d'avoir une première base stable et commune en mettant au propre notre diagramme de classe et en implémentant les premières fonctionnalités essentielles de notre projet, ce dernier était lancé. Des réunions hebdomadaires, nous permettant d'échanger sur nos difficultés rencontrées ou encore sur des incertitudes concernant des choix d'implémentations, nous ont été d'une grande aide pour avancer à un rythme régulier dans la réalisation de notre projet et surtout dans l'optique de pouvoir fournir un livrable fonctionnel avant l'échéance.
De plus, nous avons appliqué à notre base de code un linting uniforme, afin d'éviter des erreurs de styles ou de programmation, et nous nous sommes appliqués à laisser des commentaires utiles, pour favoriser une meilleure lisibilité de notre code entre nous, et dans le futur.
L'acquisition de bonnes pratiques agiles nous a donc permis de livrer au bout de quatre semaines un programme répondant aux différentes attentes formulées dans le sujet.

[1] *Lorsqu'on parle de la méthode agile SCRUM on sous-entend une idée de rythme en continu. Si l'on souhaite visualiser ce concept à l'aide d'un schéma, on retrouvera à chaque fois différentes boucles, dont la plus imposante représente une itération qui correspond à un Sprint.*

4. Modélisation UML

(Justification de la structure en arbre, des héritages, des raisons nous ayant poussé à séparer la partie binaire et unaire et explications du diagramme de classe)

Diagramme de classe



Comme on peut le voir sur le diagramme de classe ci-dessus, la classe abstraite principale, au cœur de notre projet est la classe: **Gate.**

Cette classe **abstraite** est redéfinie en plusieurs classes **concrètes** qui sont: *InputGate*, *XorGate*, *NXorGate*, *OrGate*, *NorGate*, *AndGate* et *NAndGate*. Un Gate représente donc une porte logique qui peut être soit unaire, c'est-à-dire n'ayant à entrée unique, soit binaire, donc à deux entrées. Une dernière différence peut être faite entre les *InputGates*, qui représentent les feuilles de l'arbre booléen, et les classes *BinaryGate* et *UnaryGate* qui ne peuvent qu'être instanciées à travers leurs classes filles.
Finalement, *l'OutputGate* n'étant pas une classe concrète de *Gate*, mais plutôt une classe à part, qui utilise *Gate* en manipulant des vecteurs de vecteurs de

pointeurs vers des instances de celle-ci, est donc reliée à la classe abstraite *Gate* par une association simple. La classe OutputGate représente non seulement la sortie du circuit combinatoire, mais d'une certaine manière aussi le circuit tout entier, puisque la classe contient toutes les informations sur les portes logiques composant le circuit en question. Cette propriété de l'OutputGate permet ainsi à l'utilisateur d'interagir avec le circuit combinatoire en appliquant diverses méthodes à ce dernier

Différents choix de modélisation appliqués au diagramme de classe UML ci-dessus, méritent une justification plus poussée, comme notamment le fait d'avoir séparé les portes à deux entrées des portes à une seule entrée. Il aurait en effet été légitime de regrouper ces deux sous-classes en une seule à l'aide de la surcharge du constructeur, compte tenu du nombre limité de portes logiques à implémenter. Notre décision finale s'est cependant portée sur une vision plus générale du projet et sur deux aspects primordiaux de la programmation **SOLID** : la flexibilité et la maintenabilité du code. Lorsqu'on écrit du code, il est en effet très important de faciliter des modifications futures et de permettre des mises à jour rapides de ce dernier.

Dans cette optique, nous avons préféré séparer les deux classes *BinaryGate* et *UnaryGate* afin de permettre un ajout simple de nouvelles portes logiques et une organisation évidente des différentes classes filles. Si par exemple le projet était amené à grandir de manière à traiter non pas seulement les entrées de variables booléennes, mais aussi plus généralement des variables représentant des nombres binaires, faire une distinction des portes logiques selon leur arité se révèlera bien pratique.

Finalement, nous avons décidé de mettre les constructeurs de ces deux classes en '*protected*' afin d'interdire l'utilisation de ces derniers en dehors de leurs classes filles, c'est-à-dire éviter toutes instanciations publiques d'objets BinaryGate ou UnaryGate sans devoir passer par leurs classes filles. Une telle conception implique que la création des différentes portes logiques nécessitera l'usage explicite de la classe. Par ailleurs, les constructeurs par copie des différentes portes logiques sont également protégés, voire même privés, afin d'éviter un quelconque danger suite à une mauvaise manipulation d'une copie d'un objet. Nous avons par contre choisi de mettre en publique le constructeur par copie de l'OutputGate, afin de laisser à l'utilisateur la liberté de modifier les entrées de son circuit combinatoire, tout en ayant encore accès à la version initiale de ce dernier.

Il s'agit donc visiblement d'une structure de polymorphisme, qui se traduit par des méthodes dont l'appel déclenche l'exécution d'une méthode différant en fonction du type de l'objet à l'origine de cet appel.

5. Implémentation C++

(Présentation détaillée des éléments d'implémentation, des différentes méthodes importantes et des choix techniques réalisés)

Dans cette partie nous allons évoquer et expliquer les différentes méthodes, que nous avons implémentées dans les différentes classes décrites ci-dessus afin d'aboutir à un résultat satisfaisant les différentes demandes de notre *client*.

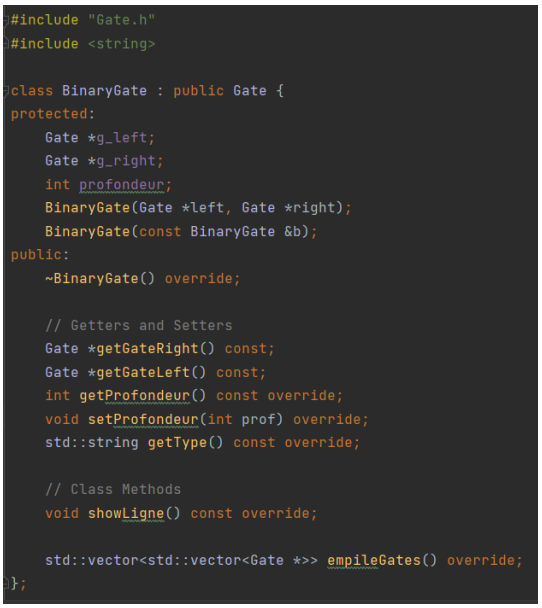
- **La classe abstraite *Gate***

```
class Gate {
protected:
    Gate();
    Gate(const Gate &g);
public:
    virtual ~Gate();

    // Getters
    virtual int getProfondeur() const = 0;
    virtual void setProfondeur(int prof) = 0;
    virtual std::string getType() const = 0;

    // Class Methods
    virtual void afficher() const = 0;
    virtual std::string getText() const = 0;
    virtual void showLigne() const = 0;
    virtual int calculer() const = 0;
    virtual void showExpression() const = 0;
    virtual std::vector<std::vector<Gate *>> empileGates() = 0;
};
```

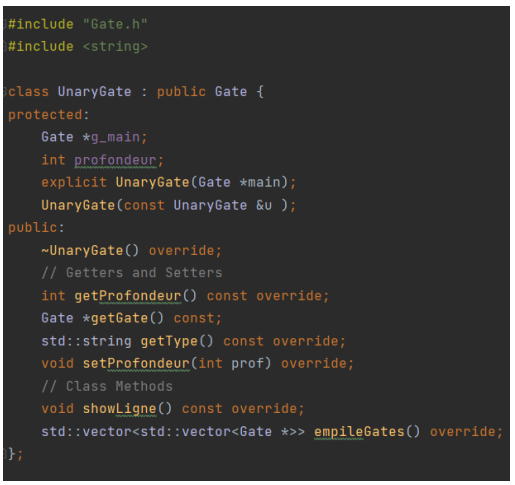
Les différentes méthodes de cette classe sont des méthodes virtuelles pures qui transforment la classe Gate en classe abstraite. En effet, dès qu'une classe possède une méthode virtuelle pure celle-ci ne pourra plus être directement instanciée. Les différentes méthodes affichées ci-dessus seront donc implémentées dans les classes filles concernées et seront d'une grande importance dans le bon déroulement des différents tests que l'on retrouve dans le main du projet: alors que la méthode *calculer*() permet de retrouver la valeur de vérité d'une expression booléenne, les méthodes *getText*() et *afficher*() vont permettre de retrouver la forme textuelle du circuit combinatoire et de l'afficher par la suite.
Par ailleurs on remarque que la majorité des méthodes sont déclarées en const, c'est-à-dire qu'elles ne modifient pas la valeur des attributs qui y sont utilisés et qu'elles pourront être utilisées sur des objets déclarés const.
Finalement, puisque la classe Gate est abstraite et ne sera jamais instanciée en tant que telle mais à travers une classe fille, la déclaration de variables dans

celle-ci n'est pas utile, et donc l'ajout d'un constructeur serait superficiel puisqu'il serait sans paramètres.

- **Les classes *BinaryGate* et *UnaryGate***



```cpp
#include "Gate.h"
#include <string>

class BinaryGate : public Gate {
protected:
    Gate *g_left;
    Gate *g_right;
    int profondeur;
    BinaryGate(Gate *left, Gate *right);
    BinaryGate(const BinaryGate &b);
public:
    ~BinaryGate() override;

    // Getters and Setters
    Gate *getGateRight() const;
    Gate *getGateLeft() const;
    int getProfondeur() const override;
    void setProfondeur(int prof) override;
    std::string getType() const override;

    // Class Methods
    void showLigne() const override;

    std::vector<std::vector<Gate *>> empileGates() override;
};
```



```cpp
#include "Gate.h"
#include <string>

class UnaryGate : public Gate {
protected:
    Gate *g_main;
    int profondeur;
    explicit UnaryGate(Gate *main);
    UnaryGate(const UnaryGate &u );
public:
    ~UnaryGate() override;
    // Getters and Setters
    int getProfondeur() const override;
    Gate *getGate() const;
    std::string getType() const override;
    void setProfondeur(int prof) override;
    // Class Methods
    void showLigne() const override;
    std::vector<std::vector<Gate *>> empileGates() override;
};
```

Les constructeurs des classes *BinaryGate* et UnaryGate sont *protected* puisque ces deux classes ne seront instanciées qu'à travers leurs classes filles. Ainsi il est suffisant de limiter la portée à 'protected' afin d'autoriser uniquement aux classes filles l'utilisation de ces constructeurs.
Dans chacune des deux classes (et de leurs classes filles évidemment) on stocke le pointeur vers le Gate, qui a été passé au constructeur comme paramètre, afin de pouvoir y accéder dans les diverses méthodes de classe. On remarque ici la différence évidente entre le constructeur d'un *UnaryGate* et d'un *BinaryGate*, dont l'un ne prend comme paramètre qu'un seul Gate contrairement au second qui en prend deux en entrée : la porte logique droite et la porte logique gauche.
La méthode *showLigne*() sera utilisée dans l'affichage de la structure du circuit combinatoire qui sera expliquée par la suite.
La méthode la plus importante de cette partie est sans doute la méthode *empileGates*() qui est implémentée dans les trois sous-classes de Gate, à savoir *BinaryGate*, *UnaryGate* ainsi qu'*InputGate*. L'objectif principal de cette méthode est de retrouver à partir d'un OutputGate les différents nœuds composants de ce dernier, c'est-à-dire les différents Gates qui ont été utilisés pour créer l'OutputGate en question, et de les stocker dans un vecteur de vecteurs de pointeurs de Gate. Illustrons cette idée par un exemple :

A = *xor(or(a, b), and(c, d)*

→L'*OutputGate* = A
→Les *BinaryGate* : *OrGate(a,b)*, *AndGate(c,d)*, *XorGate(OrGate(a,b), AndGate(c,d))*
→Les *InputGate* : *InputGate(a)*, *InputGate(b)*

Les pointeurs de Gates seront stockés de façon à ce que toutes les portes qui se trouvent à la même profondeur du circuit combinatoire, soit ajoutées au même vecteur de pointeurs de Gate qui lui même sera ajouté au vecteur de vecteurs final. Une exception sera pourtant faite pour les InputGate. Malgré le fait que les InputGate ont par défaut la profondeur 0, puisqu'il s'agit des feuilles de l'arbre, elles seront ajoutées au vecteur qui contiendra les éléments directement voisins de ces derniers. L'élément en question ne se trouvera donc pas forcément dans le sous-vecteur qui représente les éléments à la racine de l'arbre. Pour pallier un quelconque souci d'affichage, l'ajout d'InputGate à un vecteur autre que le sous-vecteur des éléments à la racine de l'arbre, entraînera l'ajout de 2 nullptr supplémentaires afin de mieux gérer l'espace entre les portes logiques.
Reprenons l'exemple ci-dessus. Le vecteur sera alors de la forme (en ignorant les nullptr) :

```
[[*InputGate(a),*InputGate(b),*InputGate(c),*InputGate(d)][*OrGate(a,b),*AndGate(c,d)]
[*XorGate(OrGate(a,b), *AndGate(c,d))]]
```

Créer une telle file des différentes portes logiques permettra de faciliter la réalisation de l'affichage sous forme de circuits de *l'OutputGate* en question (dans l'exemple ci-dessus A).
Pour développer une méthode qui permette d'obtenir le vecteur souhaité à partir d'un *OutputGate*, il a été important de distinguer dans un premier temps les 3 cas possibles : *l'InputGate*, le *BinaryGate* et le *UnaryGate.*

- **1er cas - InputGate:**

Le cas de l'InputGate est le plus simple. Il suffit en effet de créer un sous-vecteur contenant le pointeur vers l'InputGate courant (this), et de l'ajouter au vecteur de vecteur de pointeurs final.

- **2e cas - UnaryGate:**

Lorsque le Gate auquel on applique la méthode *empileGates*() est un *UnaryGate*, on commence par mettre le premier *Gate* dans un sous-vecteur, qui lui-même sera ajouté au vecteur de vecteurs de pointeur de *Gates*. Par ailleurs, on crée une file **FIFO** (First In First Out), dans laquelle on insère petit à petit les nouveaux Gates rencontrés et qui permet ainsi de garantir une bonne gestion des portes logiques. Une fois qu'un *Gate* présent dans la file est ajouté au sous-vecteur, le dernier élément de la queue est supprimé de celle-ci en utilisant le principe FIFO.
Tant qu'on a pas atteint la dernière feuille de l'arbre logique, on répète le procédé suivant :

- si entre temps les Gates ne se trouvent plus au même étage, c'est-à-dire si la profondeur a changé alors on ajoute le sous-vecteur au vecteur principal avant de le vider. Ensuite on vérifie le type du Gate, c'est-à-dire s' il s'agit d'un *BinaryGate*, *d'un InputGate* ou d'un *UnaryGate* en utilisant la fonction *getType*(). S'il s'agit d'une porte logique binaire ou bien d'une feuille, alors on applique la fonction empileGates() au Gate courant, qui va ainsi faire appel à la fonction correspondante dans la classe BinaryGates, respectivement InputGate.

- Sinon on effectue un downcast pour obtenir une instance du Gate composant l'UnaryGate courant, qui sera insérée dans la file ainsi que dans le sous-vecteur.

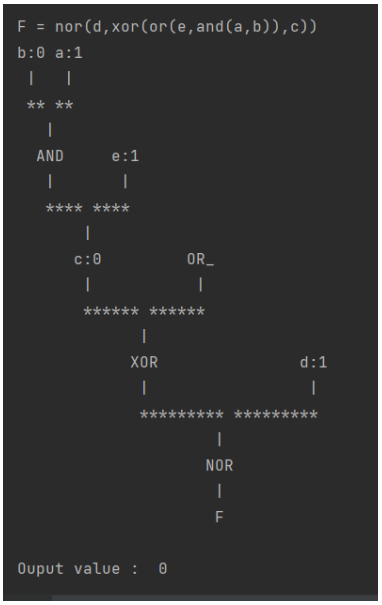
- **3e cas - BinaryGate:**

Lorsque le Gate auquel on applique la méthode empileGates() est un BinaryGate, les choses se compliquent un peu. Il ne suffira plus de comparer la nature du Gate courant, ainsi que le changement de profondeur, mais une difficulté supplémentaire s'ajoute à l'extraction des différentes portes logiques : la comparaison des longueurs des branches des deux entrées de la porte logique. Il est en effet primordial de réarranger correctement dans la file les différentes portes logiques, afin qu'aucune d'entre elles ne soit oubliée. Pour éviter qu'on arrête la recherche des portes logiques prématurément, c'est-à-dire à la première feuille que l'on trouve on ajoute une boucle supplémentaire :

```
do {
    if (currentGate->getProfondeur() == 0) {
        sous_pile.push_back(nullptr);
        sous_pile.push_back(nullptr);
    }
    g_queue.pop();
    currentGate = g_queue.front();
} while (currentGate->getProfondeur() == 0 && (g_queue.size() > 1));
```

Ces quelques lignes permettent de contourner le problème décrit ci-dessus. Par ailleurs, les nullptr que l'on ajoute au sous-vecteur vont permettre d'afficher correctement le graphe en laissant l'espace nécessaire au-dessus des InputGate.

Finalement, afin de faciliter l'affichage sous forme d'une structure d'arbre booléen, on a décidé d'ordonner le vecteur de façon à mettre les feuilles catégoriquement à droite dans les sous-vecteurs, ce qui permet d'éviter une incohérence comme sur la capture d'écran à droite.



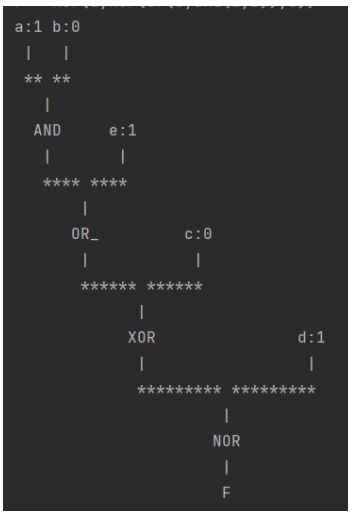
On remarque clairement que le OR et le c:0 sont inversés, ce qui cause une incohérence dans l'affichage de l'arbre logique.

Afin de contourner une telle erreur on ajoute les lignes suivantes au code de la méthode :

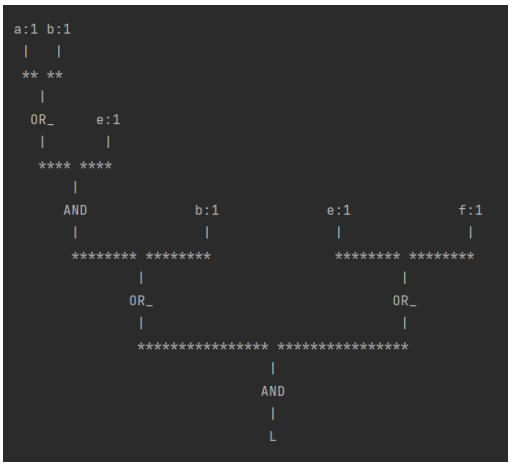
```
if ((((BinaryGate *) (currentGate))->getGateRight()->getProfondeur()) >
    (((BinaryGate *) (currentGate))->getGateLeft()->getProfondeur())) {
    g_queue.push(((BinaryGate *) (currentGate))->getGateRight());
    sous_pile.push_back(g_queue.back());
    g_queue.push(((BinaryGate *) (currentGate))->getGateLeft());
  if ((((BinaryGate *) (currentGate))->getGateLeft()->getProfondeur())!=0){
((BinaryGate  *)(currentGate))->getGateLeft()->setProfondeur(((BinaryGate  *)
(currentGate))->getGateRight()->getProfondeur());
    }
    sous_pile.push_back(g_queue.back());
} else {
```

```
    g_queue.push(((BinaryGate *) (currentGate))->getGateLeft());
    sous_pile.push_back(g_queue.back());
    g_queue.push(((BinaryGate *) (currentGate))->getGateRight());
    sous_pile.push_back(g_queue.back());
 if ((((BinaryGate *) (currentGate))->getGateRight()->getProfondeur())!=0){
((BinaryGate *) (currentGate))->getGateRight()->setProfondeur(((BinaryGate *)
(currentGate))->getGateLeft()->getProfondeur());
    }
}
```

Cela va nous permettre d'obtenir un affichage final cohérent comme le suivant :



Les deux if dans les lignes précédentes permettent de gérer le cas suivant :



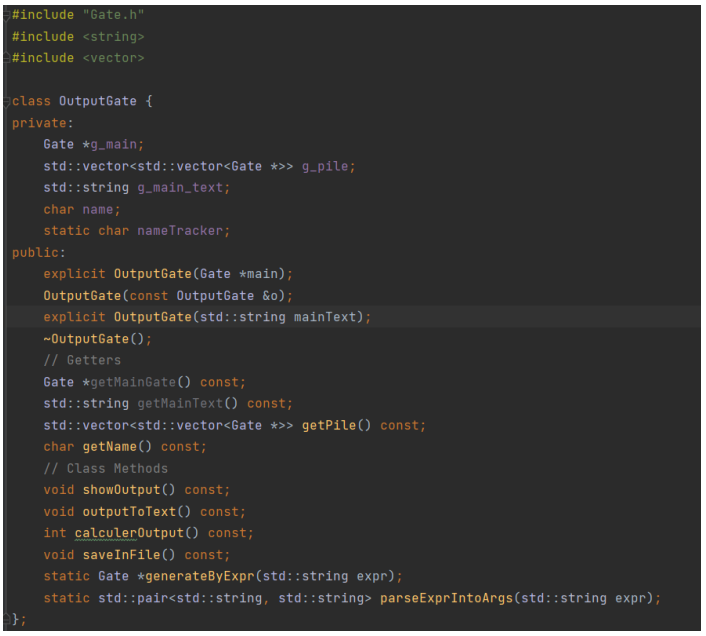
On voit ici que la profondeur théorique du OrGate à droite devrait être 1, en partant du principe que tous les InputGate sont de profondeur 0. Cependant on souhaiterait l'afficher à la profondeur 3 et gérer ses fils, donc les InputGate e et f comme tel. Ce sont les deux if que l'on retrouve dans les lignes présentées précédemment qui nous le permettent.

- **La classe *InputGate***

```cpp
#include "OutputGate.h"
#include <string>

class InputGate : public Gate {
private:
    char const name;
    bool value = false;
    int profondeur = 0;
    InputGate(const InputGate &i );

public:
    explicit InputGate(char varName, bool value = false);
    ~InputGate() override;
    // Getters and Setters
    char getName() const;
    int getValue() const;
    int getProfondeur() const override;
    std::string getType() const override;
    void setValue(bool newValue);
    void setProfondeur(int prof) override;
    std::string getText() const override;

    // Class Methods
    void afficher() const override;
    int calculer() const override;
    void showExpression() const override;
    void showLigne() const override;
    std::vector<std::vector<Gate *>> empileGates() override;
};
```

Comme on vient de le voir, *InputGate* est une classe fille de la classe abstraite *Gate*, qui implémente donc les méthodes virtuelles pures de celle-ci. Mise à part de simples Getters ou encore la méthode empileGates(), la classe InputGate contient des méthodes primordiales au bon fonctionnement du polymorphisme d'héritage: *calculer*, *afficher*() et *getProfondeur*(). Malgré que ces méthodes semblent très basiques, en ne renvoyant qu'une seule valeur à chaque fois, elles ne le sont pas pour l'arbre booléen. Dans chaque classe fille de la classe abstraite Gate, les méthodes *getProfondeur*() ainsi que *calculer*() et *afficher*() vont faire des appels récursifs sur leurs composants, jusqu'à atteindre le **cas terminal** qui se trouve justement dans la classe InputGate, d'où la grande responsabilité de cette dernière pour faire fonctionner le polymorphisme d'héritage.
Finalement, la méthode *setValue*() de la classe InputGate() va permettre à l'utilisateur de modifier la valeur de vérité d'une instance d'InputGate qui est utilisée dans un circuit combinatoire, même à posteriori de la création de ce dernier. Il est également important de souligner que le constructeur de la classe prend comme paramètre un caractère représentant l'input, ainsi qu'un bool qui sera défini par défaut avec la valeur false.

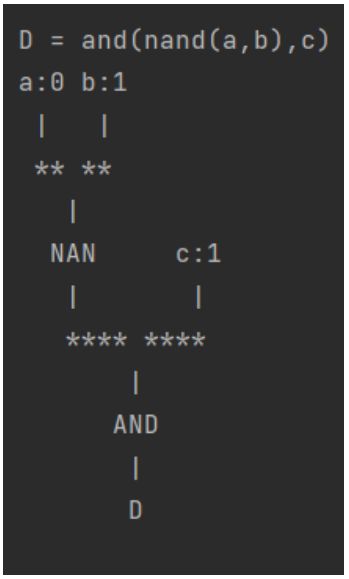
- **La classe *OutputGate***



```
#include "Gate.h"
#include <string>
#include <vector>

class OutputGate {
private:
    Gate *g_main;
    std::vector<std::vector<Gate *>> g_pile;
    std::string g_main_text;
    char name;
    static char nameTracker;
public:
    explicit OutputGate(Gate *main);
    OutputGate(const OutputGate &o);
    explicit OutputGate(std::string mainText);
    ~OutputGate();
    // Getters
    Gate *getMainGate() const;
    std::string getMainText() const;
    std::vector<std::vector<Gate *>> getPile() const;
    char getName() const;
    // Class Methods
    void showOutput() const;
    void outputToText() const;
    int calculerOutput() const;
    void saveInFile() const;
    static Gate *generateByExpr(std::string expr);
    static std::pair<std::string, std::string> parseExprIntoArgs(std::string expr);
};
```

La classe *OutputGate* représente sans doute le cœur du projet. C'est ici que seront majoritairement utilisées toutes les méthodes rencontrées auparavant afin de répondre aux demandes du client, c'est-à-dire aux directives du sujet.
Cette classe n'est **pas** une sous-classe de Gate, étant donné qu'elle a sa propre structure personnelle qui se distingue de celles des portes logiques individuelles.
Une première particularité de cette classe est son constructeur : comme on le voit ci-dessus il y a deux possibilités différentes pour instancier un objet *OutputGate* - soit en lui passant un pointeur vers un Gate, soit en lui passant une chaîne de charactères représentant soit la forme textuelle d'un Gate, soit le chemin vers un fichier texte.
Dans la classe *OutputGate* sera non seulement stocké le Gate qui lui sera passé par paramètres et sur lequel on appliquera la méthode *empileGates*(), mais également la pile de portes logiques obtenue à l'aide de cette dernière - le vecteur de vecteurs de pointeurs de Gates.
C'est cette pile qui sera utilisée dans la méthode showOutput() dont l'objectif est d'afficher un circuit combinatoire correspondant à l'instance d'OutputGate qui fait appel à la méthode en question. Pour se faire, on parcourt un par un les sous-vecteurs de la pile et on affiche sur une même ligne la forme textuelle des Gates qui s'y trouvent en utilisant la méthode *afficher*(). Les différentes portes ont une **largeur de 3 caractères**, quelque soit le type de la porte logique (InputGate, BinaryGate ou UnaryGate) et sont séparés par un nombre d'espace bien choisi : initialisé à 1 et prenant à chaque étape suivante le 2*nombre

d'espaces en début de ligne (debutLigne+=(espace_milieu / 2) + 2) augmenté de 1.
Pour rendre l'affichage du circuit combinatoire plus agréable on rajoute non seulement des traits ' | ' qui représentent les entrées d'une porte logique mais également des '*' afin de relier les différentes portes logiques. Voici un exemple d'affichage :



L'outputGate D a été instancié en utilisant le constructeur qui prend pour paramètre un Gate qui est ici un *AndGate*. Les feuilles de l'arbre logique sont les InputGate donc ici a, b et c qui ont pour valeur logique initiale respectivement 0 (= valeur par défaut d'un InputGate), 1 et 1. L'espace à chaque début de ligne est debutLigne+=(espace_milieu / 2) + 2.
.
Le nombre de '*' qui sont ajoutés entre deux portes logiques correspond à 4 + le nombre d'espaces qui les séparent -1.
Finalement la lettre D correspond au nom de l'OutputGate, qui lui est stocké à l'aide d'une variable statique et incrémenté à chaque nouvelle instanciation. Ainsi le nommage des Objets *OutputGate* se fait de façon automatique en attribuant à chaque nouvel *OutputGate* la lettre correspondant à la variable statique "*nameCracker*" incrémentée de 1.

Les méthodes *outputToText*() et *calculerOutput*() de la classe OutputGate font appel aux méthodes respectives *calculer*() et *afficher*() des différents Gates, en déclenchant l'appel récursif sur le Gate passé par paramètre au constructeur de la classe lors de l'instanciation de l'Objet.

La méthode *saveInFile*, écrit dans un fichier texte toutes les informations intéressantes à propos d'un circuit. Pour cela, on redirige le flux *cout* de la sortie standard vers un nouveau fichier texte. On envoie ensuite vers *cout* de quoi relire le fichier, on appelle *outputToText* ainsi que *showOutput*, dont les *cout* écriront directement dans le fichier, puis on rétablit le flux vers la sortie standard et on ferme le fichier.

Enfin, pour créer un circuit à partir d'une expression textuelle ou d'un fichier (en trouvant l'expression textuelle dans le fichier et en l'utilisant), on a habilement

surchargé le constructeur d'OutputGate pour qu'il accepte en argument une chaîne de caractère.
Cette chaîne de caractère peut représenter deux choses. Soit elle représente un chemin vers un fichier texte, soit une expression textuelle de circuit booléen. Si on est dans le premier cas (que l'on repère en vérifiant si la chaîne se termine par ".txt"), on ouvre le fichier et on en extrait l'expression textuelle.
Pour ensuite générer le circuit à partir de l'expression textuelle, on fait appel à une fonction statique récursive:

Cette fonction, *generateByExpr*, crée et renvoie récursivement le bon type de Gate selon ce qu'elle lit dans l'expression textuelle. Son cas terminal est un InputGate. Sinon elle va parser correctement l'expression pour créer la porte à gauche d'une paire de parenthèses et lui donner en argument l'appel de sa propre fonction sur la chaine de caractère parsée.
Pour parser correctement l'expression, elle fait appel à une autre fonction statique, *parseExpreIntoArgs*, qui détecte si on a affaire à une porte unaire ou binaire, et renvoie un objet pair contenant le ou les membres à l'intérieur des parenthèses de l'expression textuelles, pour les donner en argument à *generateByExpr*.

6. Petite rétrospective de projet

En aval de tout projet, il est important de porter un regard critique sur le travail fourni afin de mettre en évidence les différentes problématiques rencontrées lors de la réalisation de ce dernier, ainsi que des améliorations réalistes à appliquer au livrable, dans l'optique de ne plus rencontrer les mêmes difficultés dans des projets futurs.

Difficultés rencontrées :
Il est normal qu'un projet ne se passe pas toujours de façon idéale et sans aucun inconvénient. Mais se rendre compte des difficultés rencontrées et essayer d'y faire face du mieux qu'on peut est une chose très importante qui permet de concrétiser tout travail.
Il va sans dire que nous avons pu rencontrer diverses problématiques ou complications tout au long de la réalisation du projet, comme notamment le calcul des espaces pour rendre l'affichage du circuit combinatoire agréable. Mais également la recherche des feuilles de l'arbre booléen dans la méthode empileGates pour ne pas s'arrêter prématurément dans la recherche des portes logiques composant l'OutputGate, la gestion des différents cas d'erreurs lors d'une mauvaise entrée textuelle passée comme paramètre au constructeur d'OutputGate, la création récursive des bonnes portes logiques pour la création d'un circuit à partir d'une expression textuelle, ainsi que le travail collaboratif à distance à gérer avec les examens terminaux.
Les différentes difficultés rencontrées ont donc été principalement soit d'ordre technique, soit d'un point de vue gestion de projet. Deux types de problématiques qui ont demandé une communication explicite entre les membres du binôme de projet ainsi qu'une bonne organisation et un travail régulier.

Améliorations possibles :
Même si le livrable final est fonctionnel et répond aux divers besoins d'un client, on peut toujours trouver des améliorations à faire sur un projet, chose qui concerne également notre circuit combinatoire. Ainsi, on pourrait penser ici à un affichage amélioré de ce dernier, à savoir sans répétition des InputGates qui apparaissent plusieurs fois dans une même expression booléenne. Il faudrait également dans un second temps, refactoriser les fonctions qui permettent l'affichage, afin de rendre le code plus lisible et facilement modifiable.

Par ailleurs on pourrait également penser à permettre la détection d'expression booléenne dans un texte quelconque, indépendamment du nombre d'espaces qui séparent deux mots qui entourent l'expression en question.
Une autre amélioration possible serait de pouvoir traiter et supporter des circuits bien plus complexes, c'est-à-dire de pouvoir par exemple facilement générer des circuits plus compliqués, dans l'optique de les combiner. (additionneur, multiplexeur, etc)

7. Conclusion

D'un point de vue global, malgré les difficultés que nous avons pu rencontrer, nous sommes satisfaits du livrable que nous avons pu fournir au bout de 4 semaines, étant donné que les différentes fonctionnalités implémentées renvoient bien le résultat souhaité et que toutes les demandes formulées dans le sujet ont été traitées soigneusement. Par ailleurs nous sommes particulièrement fiers d'avoir mis en place une organisation de projet efficace, qui nous a permis d'avancer rapidement malgré la charge de travail des examens finaux, la période des fêtes, et le covid d'un des membres du binôme.

De plus, c'était fortement enrichissant de mettre en œuvre de manière pratique, des concepts que l'on avait abordés théoriquement quelques semaines auparavant dans le cours de Complexité, tout en en profitant pour approfondir nos compétences à la fois en programmation et en bonnes pratiques de code et de gestion de projet.

En vous remerciant pour votre temps.

Madeleine et Quentin