[Aller plus loin] L'algèbre de Boole

Vous avez vu dans les chapitres précédents comment utiliser les booléens et les opérations de base : ET, OU, NON et OU-EXCLUSIF. En fait, l'utilisation des booléens est plus riche et complexe que cela. Ils sont à la base d'une branche des mathématiques, appelée Algèbre de Boole, en l'honneur de son créateur, le mathématicien George Boole.

Avoir des bonnes bases avec cet algèbre est important pour tous les développeurs, et plus encore pour les développeurs C++, c'est pourquoi nous allons voir cela plus en détail.

Quelques rappels

Pour rappel, un booléen est une variable logique, qui peut prendre deux valeurs. Peu importe comment sont nommées ces valeurs, vous rencontrez parfois : “vrai” et “faux”, “oui” et “non”, “haut” et “bas”, “positif” et “négatif”, 0 et 1, “Titi” et “Grosminet”, etc.

Le C++ utilise les mots-clés `true` (*vrai*) et `false` (*faux*) pour représenter les booléens.

Certaines valeurs dans la liste ont un sens historique. En particulier, “vrai” et “faux” correspond au fait que les booléens représentent le résultat d'une proposition logique. Par exemple “la terre est plus grosse que le soleil”.

En C++, une proposition logique (que l'on appelle aussi expression logique) sera écrite en utilisant les opérateurs logiques (que vous avez déjà vu) :

Opérateur	Synonyme	Opérateur C++
ET	Conjonction	`&&`
OU	Disjonction	`||`

NON	Négation	`!`

Pour rappel, voici la table de vérité de ces opérateurs (elle est redonnée pour vous éviter de retourner dans le chapitre Logique binaire et calcul booléen, mais il faudra la connaître par cœur par la suite. Mais pas d'inquiétude, vous la connaîtrez à force de l'utiliser) :

a	b	!a	a && b	a || b
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Il est classique de définir d'autres opérateurs logiques par combinaison des opérateurs de base. Par exemple :

- NON-OU (*NOR*) correspond à `!(a || b)` ;
- NON-ET (*NAND*) correspond à `!(a && b)` ;
- OU-Exclusif (*XOR*) correspond à `(a && !b) || (!a && b)` ;
- NON-OU-Exclusif (*XNOR*) correspond à `!( (a && !b) || (!a && b) )`.

Ces opérateurs n'ont pas d'équivalent direct en C++, mais il est facile de les écrire en utilisant les opérateurs de base du C++.

Propriétés des opérateurs logiques

L'algèbre de Boole définit un certain nombre de propriétés. Il est assez facile d'écrire un code C++ pour les vérifier. Un code d'exemple est donné pour la première propriété, vous devrez écrire les codes C++ correspondant aux autres propriétés comme exercice.

Ordre d'évaluation

Une expression contenant plusieurs opérateurs (et donc plus de deux opérandes) est évaluée en évaluant chaque opérateur un par un. Dans le

cas des opérateurs logiques, les opérateurs sont évalués en suivant les règles suivantes :

- les opérateurs sont évalués dans l'ordre : NÉGATION, ET puis OU ;
- lorsque les opérateurs sont identiques, ils sont évalués de gauche à droite.

Par exemple :

```
a || b && c
```

pourra s'écrire de la façon suivante :

```
a || (b && c)
```

Un autre exemple :

```
a || b || c
```

pourra s'écrire de la façon suivante :

```
(a || b) || c
```

Ces règles s'appliquent aux opérateurs logiques. Vous avez déjà vu que les opérateurs arithmétiques ont aussi un ordre d'évaluation spécifique (multiplication et division avant addition et soustraction). Vous verrez par la suite les règles d'évaluation pour l'ensemble des opérateurs existant en C++.

Dans tous les cas, il est plus simple et moins ambiguë d'utiliser les parenthèses pour exprimer clairement une expression.

Exercice : évaluer l'ordre des opérations suivantes, en ajoutant les parenthèses pour isoler chaque opérateur, sans changer le résultat de l'expression.

```
a && b && c
a || b || c
a || b && c
```

```
a && b || c

!a && b
a || !b

a && b || c && d
a || b && c || d
```

Complémentarité

Dans certains cas, l'utilisation de l'opérateur négation peut se simplifier. Le cas le plus simple est la double-négation qui donne le même résultat :

```
!!a == a
```

Sous forme de table de vérité :

a	!a	!!a
0	1	0
1	0	1

Lorsque l'opérateur négation est utilisé avec d'autres opérateurs, on peut également simplifier dans certains cas :

```
a && !a == false
a || !a == true
```

Sous forme de table de vérité :

a	!a	a && !a	a || !a
0	1	0	1
1	0	0	1

Idempotence

La première notion est relativement simple : appliquer un opérateur en utilisant deux fois la même valeur est équivalent à utiliser directement la

variable. Si on regarde la table de vérité précédente, on peut remarquer que lorsque `a` et `b` sont égaux, alors le résultat des opérations logiques aura la même valeur :

a	b	a && b	a || b
0	**0**	**0**	**0**
0	1	0	1
1	0	0	1
1	**1**	**1**	**1**

De façon plus formelle, on peut écrire :

$$ \text{A} ~ \text{ET} ~ \text{A} ~ = ~ \text{A} $$ $$ \text{A} ~ \text{OU} ~ \text{A} ~ = ~ \text{A} $$

Eléments neutres

Un élément neutre est une valeur booléenne qui ne va pas changer le résultat d'une expression logique. Pour l'opérateur ET, on peut remarquer dans la table de vérité que si `a` est vrai, le résultat de l'expression aura la même valeur que `b` :

a	b	a && b
0	0	0
0	1	0
1	**0**	**0**
1	**1**	**1**

Pour l'opérateur OU, ça sera faux qui est l'élément neutre :

a	b	a || b
0	**0**	**0**
0	**1**	**1**
1	0	1
1	1	1

On peut écrire :

$$ \text{A} ~ \text{ET} ~ \text{Vrai} ~ = ~ \text{A} $$ $$ \text{A} ~ \text{OU} ~ \text{Faux} ~ = ~ \text{A} $$

Ces relations permettent de simplifier une expression. Si vous démontrez qu'une opérande est toujours vraie ou toujours fausse, vous pouvez simplifier certaines expressions en supprimant l'opérateur et l'opérande.

Absorption

L'absorption peut être considérée comme l'inverse de l'élément neutre : si `a` est faux, le résultat de l'opérateur ET sera toujours faux.

a	b	a && b
0	**0**	**0**
0	**1**	**0**
1	0	0
1	1	1

Pour l'opérateur OU, si `a` est vrai, le résultat sera toujours vrai :

a	b	a || b
0	0	0
0	1	1
1	**0**	**1**
1	**1**	**1**

L'écriture formelle :

$$ \text{A} ~ \text{ET} ~ \text{Faux} ~ = ~ \text{Faux} $$ $$ \text{A} ~ \text{OU} ~ \text{Vrai} ~ = ~ \text{Vrai} $$

Cette propriété est particulièrement intéressante en C++, puisque cela permet de ne pas évaluer une expression, si possible. Comme expliqué dans le chapitre Logique binaire et calcul booléen, le C++ utilise la *lazy evaluation* (évaluation paresseuse) lorsque c'est possible. Pour évaluer les expressions suivantes :

$$ \text{A} ~ \text{ET} ~ \text{Expression complexe} $$ $$ \text{B} ~ \text{OU} ~ \text{Expression complexe} $$

Si `A` est *faux*, le résultat sera toujours *faux*, quelle que soit la valeur de l'expression complexe. Le programme C++ va donc évaluer la valeur de `A` dans un premier temps. Si cette valeur est *faux*, le programme n'évaluera pas l'expression complexe et retournera directement *faux*. De même, si `B` est *vrai*, le programme n'évaluera pas l'expression complexe et retournera directement *vrai*.

Commutativité

La commutativité signifie que l'on peut changer l'ordre des valeurs (opérandes) dans une expression, sans changer le résultat. Cette propriété est valide pour les opérateurs binaires (qui prennent deux opérandes, donc les opérateurs ET et OU), mais pas pour NON (opérateur unaire, qui ne prend qu'une opérande).

En utilisant une écriture plus formelle, on peut donc écrire :

$$ \text{A} ~ \text{ET} ~ \text{B} \iff \text{B} ~ \text{ET} ~ \text{A} $$ $$ \text{A} ~ \text{OU} ~ \text{B} \iff \text{B} ~ \text{OU} ~ \text{A} $$

On peut vérifier cela en écrivant la table de vérité correspondant aux deux expressions :

a	b	a && b	b && a
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Vérifions cela avec un code C++ :

main.cpp

```
#include <iostream>
```

```
int main() {
    std::cout << "(A ET B) est équivalent à (B ET A), pour A
faux et B faux ? " <<
        std::boolalpha << ((false && false) == (false &&
false)) << std::endl;
        //                    A   ET   B          B   ET   A
}
```

affiche :

```
(A ET B) est équivalent à (B ET A), pour A faux et B faux ?
true
```

Exercice : écrire les trois autres lignes, correspondant à A vrai et B faux, à A faux et B vrai et à A vrai et B vrai.

Associativité

Comme vu précédemment, les opérateurs identiques sont évalués de gauche à droite. En fait, les opérateurs logiques sont associatifs et l'ordre d'évaluation ne change pas le résultat :

```
(a && b) && c == a && (b && c) == a && b && c

(a || b) || c == a || (b || c) == a || b || c
```

Exercice : compléter les tables de vérité suivantes :

a	b	c	a && b	(a && b) && c	b && c	a && (b && c)
0	0	0	…	…	…	…
…	…	…	…	…	…	…

a	b	c	a || b	(a || b) || c	b || c	a || (b || c)
0	0	0	…	…	…	…
…	…	…	…	…	…	…

Distributivité

Pour terminer avec les propriétés de base des opérateurs logiques, la distributivité fait intervenir trois opérandes et deux opérateurs différents (contrairement à l'associativité).

```
(a && b) || c == (a || c) && (b || c)

(a || b) && c == (a && c) || (b && c)
```

Note : il est classique de noter l'opérateur ET comme une multiplication et l'opérateur OU comme une addition, du fait de leur similarité. Avec cette écriture, les opérateurs logiques deviennent :

$$ (\text{A} ~ \text{OU} ~ \text{B}) ~ \text{ET} ~ \text{C} \iff (\text{A} ~ + ~ \text{B}) ~ \times ~ \text{C} $$

Sous cette forme, on reconnaît plus facilement la propriété : “le produit d'une somme est égal à la somme des produits”. Celle-ci est connue sous le nom de “distributivité de la multiplication par rapport à l'addition” et permet d'écrire :

$$ (\text{A} ~ + ~ \text{B}) ~ \times ~ \text{C} \iff \text{A} ~ \times ~ \text{C} ~ + ~ \text{B} ~ \times ~ \text{C} $$

Si on revient aux opérateurs logiques, on obtient finalement :

$$ \text{A} ~ \times ~ \text{C} ~ + ~ \text{B} ~ \times ~ \text{C} \iff (\text{A} ~ \text{ET} ~ \text{C}) ~ \text{OU} ~ (\text{B} ~ \text{ET} ~ \text{C}) $$

Ce qui correspond bien à la propriété initiale. On parle donc de “distributivité de l'opérateur ET par rapport à l'opérateur OU”. Si vous vous souvenez de la propriété pour l'addition et la multiplication, vous pouvez retrouver facilement la propriété équivalente pour les opérateurs logiques.

Une remarque importante quand même : contrairement à l'addition et la multiplication, la propriété équivalente obtenue en inversant les opérateurs est vraie pour les opérateurs logiques : l'opérateur OU est

distributif par rapport à l'opérateur ET. Ceci n'est pas vrai pour les opérateurs arithmétiques (l'addition n'est pas distributive par rapport à la multiplication).

Exercice : compléter les tables de vérité :

a	b	c	a && b	(a && b) || c	a || c	b || c	(a || c) && (b || c)
0	0	0	…	…	…	…	…
…	…	…	…	…	…	…	

a	b	c	a || b	(a || b) && c	a && c	b && c	(a && c) || (b && c)
0	0	0	…	…	…	…	…
…	…	…	…	…	…	…	

Lois de De Morgan

Les lois (ou théorèmes) de De Morgan permettent de remplacer un opérateur ET dans une expression logique par un opérateur OU et vice-versa.

```
!(a && b) == !a || !b

!(a || b) == !a && !b
```

Exercices

- Écrire les opérateurs NON-ET, NON-OU, OU-Exclusif et NON-OU-Exclusif avec les opérateurs de base du C++.
- Écrire tous les opérateurs en utilisant uniquement l'opérateur NON-ET.
- Écrire un additionneur à un bit. Un additionneur va prendre deux valeurs booléennes et retournera la somme et la retenue de ces valeurs, suivant la table de vérité suivante :

a	b	Somme	Retenue
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

faire un schéma

- Écrire un additionneur quatre bits.
- Écrire un multiplicateur 4×4 bits.
- Écrire un multiplexeur.

Chapitre précédent	Sommaire principal	Chapitre suivant