

# Tutoriel Boost.Tribool

par 3DArchi ([Page d'accueil](#))

Date de publication : 22 décembre 2008

Dernière mise à jour : 17 février 2009

Cet article présente la bibliothèque Tribool de la famille de bibliothèques Boost.




I - Révisions.....	3
II - Avant-propos.....	3
III - Remerciements.....	3
IV - Introduction.....	3
IV-A - Pré-requis.....	3
IV-B - Objectifs.....	3
IV-C - Convention.....	3
V - Premiers pas.....	4
V-A - Une variable Boost.Tribool.....	4
V-B - La valeur indéterminée.....	4
V-C - Tester la valeur.....	4
V-D - Comparaison n'est pas raison.....	5
V-E - Accéder à la valeur.....	7
V-F - Ecrire une variable Boost.Tribool vers un flux de sortie.....	7
V-G - Lire une variable Boost.Tribool depuis un flux d'entrée.....	8
VI - Utilisation de Boost.Tribool.....	8
VI-A - En-têtes.....	8
VI-B - Espace de nommage.....	8
VI-C - Le type boost::logic::tribool.....	9
VI-C-1 - La classe.....	9
VI-C-2 - Type 'indéterminé'.....	9
VI-C-3 - Opérateur !.....	10
VI-C-4 - Opérateur &&.....	10
VI-C-5 - Opérateur   .....	11
VI-C-6 - Opérateur ==.....	12
VI-C-7 - Opérateur !=.....	12
VI-D - Les flux et boost::logic::tribool.....	13
VI-D-1 - Les valeurs.....	13
VI-D-2 - Entrée.....	14
VI-D-3 - Sorties.....	15
VI-D-4 - Facette pour l'état 'indéterminé'.....	16
VI-D-5 - STL sans implémentation de locale.....	17
VI-E - En résumé.....	17
VII - Comment ça marche ?.....	18
VII-A - Les fichiers.....	18
VII-B - La classe.....	18
VII-C - Le type 'indéterminé'.....	19
VII-D - Les opérateurs.....	19
VII-E - Les flux.....	21
VII-E-1 - Les valeurs littérales.....	21
VII-E-2 - La facette.....	22
VII-E-3 - L'opérateur de sortie.....	22
VII-E-4 - L'opérateur d'entrée.....	23
VIII - Soyons logique !.....	26
VIII-A - Rappel sur l'algèbre de Boole.....	26
VIII-B - L'algèbre tri-booléenne.....	27
VIII-B-1 - L'opérateur ET.....	27
VIII-B-2 - L'opérateur OU.....	28
VIII-B-3 - L'opérateur NON.....	28
VIII-B-4 - L'opérateur ==.....	29
VIII-B-5 - L'opérateur !=.....	29

## I - Révisions

Version Boost	Révision article
1.36	Rédaction
1.37	Pas de modification
1.38	Pas de modification

## II - Avant-propos

Boost se veut un ensemble de bibliothèques portables destinées à faciliter la vie du développeur C++. Boost se propose de construire du code de référence pouvant aussi, à terme, être incorporé au standard (STL). Pour plus d'informations, vous pouvez consulter :

- Les différents tutoriels Boost de developpez.com :  **Tutoriels**.
- La F.A.Q Boost de developpez.com : [FAQ F.A.Q.](#)
- Le site officiel de Boost :  **Boost**.
- Le site officiel de la bibliothèque :  **Boost.Tribool**.

## III - Remerciements

Je remercie **Alp** pour ses encouragements et sa relecture, **RideKick** pour sa relecture finale et fais allégeance à **hiko-seijuro** mon parrain...

Enfin, d'une façon plus globale, je remercie les membres des forums de developpez.com qui par la qualité de leurs interventions m'ouvrent constamment de nouvelles pistes de réflexion sur ma pratique de développement.

## IV - Introduction

### IV-A - Pré-requis

Boost.Tribool est une des bibliothèques les plus simples de Boost. Aucune connaissance préalable n'est nécessaire. Le lecteur doit être familiarisé avec le langage C++. La bibliothèque s'interface avec les flux. Une connaissance minimale de ceux-ci est nécessaire, ainsi que des aspects *locale*. Le lecteur peut consulter sur le site de developpez.com les différents tutoriels sur les **flux** et sur **locale**. L'utilisation de la bibliothèque Boost.Tribool ne nécessite pas de connaissances particulières sur les autres bibliothèques Boost.

Pour une classe simple, une implémentation simple : sa compréhension ne nécessite pas un niveau de connaissance plus important que son utilisation.

### IV-B - Objectifs

Boost.Tribool, comme son nom l'indique, étend le type *bool* avec une troisième valeur. Cela permet d'avoir des variables prenant les valeurs 'vrai', 'faux' ou 'indéterminé'. Pourquoi faire compliqué alors qu'une simple énumération suffirait ? La classe offre plus qu'un simple enum en surchargeant opérateurs et fonctions adéquats.

### IV-C - Convention


On notera *true* ou *false* pour désigner ces valeurs telles qu'elles sont définies en C++ avec le type *bool*. On notera 'vrai', 'faux' ou 'indéterminé' pour les trois valeurs que peut prendre une instance de *tribool*.

## V - Premiers pas

### V-A - Une variable Boost.Tribool

La déclaration d'une variable Boost.Tribool se fait comme suit :

```
boost::logic::tribool test;
```

 *En l'absence d'initialisation explicite, la variable est initialisée par défaut à la valeur 'faux' et non 'indéterminé'.*

Pour initialiser la variable à la valeur 'vrai' ou 'faux', on se sert simplement du type booléen C++ :

```
boost::logic::tribool test;
test = true; // la variable a comme valeur 'vrai'
test = false; // la variable a comme valeur 'faux'
boost::logic::tribool vrai(true); // variable explicitement initialisée à 'vrai'
boost::logic::tribool faux(false); // variable explicitement initialisée à 'faux'
```

### V-B - La valeur indéterminée

Pour initialiser une variable Boost.Tribool à la valeur 'indéterminé', on peut utiliser le mot clé *indeterminate* proposé par la bibliothèque :

```
boost::logic::tribool test;
test = boost::logic::indeterminate;
```

On peut aussi définir son propre mot-clé pour cette valeur :

```
BOOST_TRIBOOL_THIRD_STATE(peut_etre); // définition d'un
// mot-clé spécifique pour la valeur 'indéterminé'
int main()
{
    boost::logic::tribool test;
    test = peut_etre; // utilisation de notre mot-clé
    return 1;
}
```

Attention, un mot-clé ne peut être défini localement à une fonction ou une méthode :

```
void MaFonction()
{
    BOOST_TRIBOOL_THIRD_STATE(peut_etre); // Erreur de compilation
}
void A::MaMethode()
{
    BOOST_TRIBOOL_THIRD_STATE(peut_etre); // Erreur de compilation
}
```

### V-C - Tester la valeur

'vrai', c'est vrai ! 'faux' n'est pas vrai ! 'indéterminé' ? Et bien, 'indéterminé' n'est pas vrai :

```
boost::logic::tribool test;
// ...
if(test) {
    std::cout<<"vrai!"<<std::endl;
}
```

```
else{
    std::cout<<"faux ou peut-être!"<<std::endl;
}
```

Ce qui n'est pas 'faux' est vrai ! Ce qui n'est pas 'vrai' est faux ! Ce qui n'est pas 'indéterminé' est ... indéterminé (donc n'est pas vrai) :

```
boost::logic::tribool test;
// ...
if(!test){
    std::cout<<"faux!"<<std::endl;
}
else{
    std::cout<<"vrai ou peut-être!"<<std::endl;
}
```

Enfin, pour savoir si une variable est 'indéterminé', on utilise le mot-clé comme fonction :

```
boost::logic::tribool test;
// ...
if(boost::logic::indeterminate(test)){
    std::cout<<"indéterminé!"<<std::endl;
}
else{
    std::cout<<"vrai ou faux!"<<std::endl;
}
```

Les mots-clés définis précédemment peuvent aussi être utilisés en tant que fonction :

```
BOOST_TRIBOOL_THIRD_STATE(peut_etre);
int main()
{
    boost::logic::tribool test;
    // ...
    if(peut_etre(test)){
        std::cout<<"indéterminé!"<<std::endl;
    }
    else{
        std::cout<<"vrai ou faux!"<<std::endl;
    }
    ///...
}
```

## V-D - Comparaison n'est pas raison

Essayons le code suivant :

```
boost::logic::tribool var_1(true);
boost::logic::tribool var_2(true);
if(var_1==var_2){
    std::cout<<"Jusqu'ici tout va bien"<<std::endl;
}
```

Jusqu'ici tout va bien !

Maintenant, modifions légèrement :

```
boost::logic::tribool var_1(false);
boost::logic::tribool var_2(false);
if(var_1==var_2){
    std::cout<<"Jusqu'ici tout va bien"<<std::endl;
}
```

Jusqu'ici tout va bien !

L'atterrissage :

```
boost::logic::tribool var_1(boost::logic::indeterminate);
boost::logic::tribool var_2(boost::logic::indeterminate);
if(var_1==var_2){
    std::cout<<"Jusqu'ici tout va bien"<<std::endl;
}
else{
    std::cout<<"Aïe !"<<std::endl;
}
```

Aïe !



*Il est préférable de ne pas utiliser l'opérateur '==' pour évaluer si deux variables sont dans le même état.*



*A noter que l'opérateur '!=' pour maintenir la cohérence réserve les mêmes surprises :*

```
boost::logic::tribool vrai(true);
boost::logic::tribool faux(false);
boost::logic::tribool indetermine(boost::logic::indeterminate);

if(vrai!=indetermine){
    std::cout<<"OK vrai != indéterminé"<<std::endl;
}
else{
    std::cout<<"vrai != indéterminé vaut false !"<<std::endl;
}
if(vrai!=false){
    std::cout<<"OK vrai != false"<<std::endl;
}
else{
    std::cout<<"vrai != false vaut false !"<<std::endl;
}
if(vrai!=true){
    std::cout<<"vrai != true vaut true !"<<std::endl;
}
else{
    std::cout<<"OK vrai != true vaut false"<<std::endl;
}

if(faux!=indetermine){
    std::cout<<"OK faux != indéterminé"<<std::endl;
}
else{
    std::cout<<"faux != indéterminé vaut false !"<<std::endl;
}
if(faux!=false){
    std::cout<<"faux != false vaut true !"<<std::endl;
}
else{
    std::cout<<"OK faux != false vaut false"<<std::endl;
}

if(indetermine!=boost::logic::indeterminate){
    std::cout<<"OK indéterminé != boost::logic::indeterminate"<<std::endl;
}
else{
    std::cout<<"indéterminé != boost::logic::indeterminate vaut false !"<<std::endl;
}
```

Ce code produit la sortie suivante :

```
vrai != indéterminé vaut false !
OK vrai != false
OK vrai != true vaut false
faux != indéterminé vaut false !
OK faux != false vaut false
indéterminé != boost::logic::indeterminate vaut false !
```

## V-E - Accéder à la valeur

Une variable Boost.Tribool contient un membre *value* d'un type énuméré permettant de résoudre le problème présenté ci-dessus :

```
boost::logic::tribool var;
//...
switch(var.value) {
    case boost::logic::tribool::true_value:
        std::cout<<"vrai !"<<std::endl;
        break;
    case boost::logic::tribool::false_value:
        std::cout<<"faux !"<<std::endl;
        break;
    case boost::logic::tribool::indeterminate_value:
        std::cout<<"indéterminé !"<<std::endl;
        break;
}
```

## V-F - Ecrire une variable Boost.Tribool vers un flux de sortie

L'envoi d'un booléen sur un flux produit soit une sortie numérique (0 ou 1) soit une sortie texte (par simplicité, disons *true* ou *false*) en fonction des drapeaux définis :

- *std::noboolalpha* : produit une sortie numérique ;
- *std::boolalpha* : produit une sortie sous forme de texte.

Boost.Tribool utilise ces deux drapeaux pour réaliser la sortie d'une de ses variables sous forme numérique ou sous forme textuelle :

```
std::cout<<std::noboolalpha;
std::cout<<"vrai = "<<boost::logic::tribool(true)<<std::endl;
std::cout<<"faux = "<<boost::logic::tribool(false)<<std::endl;
std::cout<<"indéterminé = "<<boost::logic::tribool(boost::logic::indeterminate)<<std::endl;

std::cout<<std::boolalpha;
std::cout<<"vrai = "<<boost::logic::tribool(true)<<std::endl;
std::cout<<"faux = "<<boost::logic::tribool(false)<<std::endl;
std::cout<<"indéterminé = "<<boost::logic::tribool(boost::logic::indeterminate)<<std::endl;
```

La sortie produite est :

```
vrai = 1
faux = 0
indéterminé = 2
vrai = true
faux = false
indéterminé = indeterminate
```

On peut personnaliser la sortie sous forme de texte pour la valeur 'indéterminé' :

```
std::locale mon_locale(// on crée une locale
    std::locale(""), // avec la locale
    new boost::logic::indeterminate_name<char>("peut-être")// c'est ici la ligne cruciale
    // on rajoute une facette qui se charge de définir le nom utilisé
);
std::cout.imbue(mon_locale);// on rajoute la facette au flux de sortie
std::cout<<std::boolalpha;
std::cout<<"indéterminé = "<<boost::logic::tribool(boost::logic::indeterminate)<<std::endl;
```

La sortie obtenue devient :

```
indéterminé = peut-être
```

## V-G - Lire une variable Boost.Tribool depuis un flux d'entrée

A l'instar de l'écriture, la lecture peut se faire avec les valeurs numériques :

```
boost::logic::tribool test;
std::stringstream("1")>>std::noboolalpha>>test; // Etat 'vrai'
std::stringstream("0")>>std::noboolalpha>>test; // Etat 'faux'
std::stringstream("2")>>std::noboolalpha>>test; // Etat 'indéterminé'
```

Si une valeur autre que 0, 1 ou 2 est lue depuis le flux, le flux est marqué en erreur : `std::ios_base::failbit` positionné :

```
boost::logic::tribool test;
std::cin>>test;
if(std::cin.fail()){
    std::cout<<"Saisir une valeur valide (0, 1 ou 2)"<<std::endl;
}
```

Et la lecture peut se faire avec les labels :

```
boost::logic::tribool test;
std::stringstream("true")>>std::boolalpha>>test; // Etat 'vrai'
std::stringstream("false")>>std::boolalpha>>test; // Etat 'faux'
std::stringstream("indeterminate")>>std::boolalpha>>test; // Etat 'indéterminé'*/
```

Un label particulier peut être défini pour la lecture :

```
std::locale mon_locale(// on crée une locale
    std::locale(""), // avec la locale
    new boost::logic::indeterminate_name<char>("peut-être") // c'est ici la ligne cruciale
    // on rajoute une facette qui se charge de définir le nom utilisé
);
boost::logic::tribool test;
std::stringstream strstr("peut-être");
strstr.imbue(mon_locale);
strstr>>std::boolalpha>>test;
```

Comme pour les valeurs numériques, si la valeur littérale n'est pas reconnue, le flux est marqué en erreur en positionnant `std::ios_base::failbit`

## VI - Utilisation de Boost.Tribool

### VI-A - En-têtes

Commençons par l'essentiel : les fichiers d'en-têtes nécessaires !

- `boost/logic/tribool.hpp`
- `boost/logic/tribool_io.hpp`

Le premier en-tête contient la définition de la classe ainsi que celles des fonctions et opérateurs surchargés. Le second fichier d'en-tête offre l'interface nécessaire à l'utilisation de cette nouvelle classe avec les entrées/sorties.

### VI-B - Espace de nommage

L'espace de nommage est :



```
namespace boost::logic
//ou
namespace boost
```

Dans ce document, nous utiliserons systématiquement l'espace de nommage complet : *boost::logic::XXX*.

## VI-C - Le type boost::logic::tribool

### VI-C-1 - La classe

```
class tribool {
public:
    enum value_t;
    // Constructeurs:
    tribool();
    tribool(bool);
    tribool(indeterminate_keyword_t);

    // Opérateur de conversion
    operator safe_bool() const;

    // valeur
    enum boost::logic::tribool::value_t value;
};
```

- tribool() :
  - **Objectif** : Constructeur par défaut.
  - **Post-condition** : La valeur prise est 'faux'.
  - **Exceptions** : Aucune.
- tribool(bool) :
  - **Objectif** : Constructeur.
  - **Post-condition** : La valeur prise est 'vrai' si le paramètre est *true*, et 'faux' si le paramètre est *false*.
  - **Exceptions** : Aucune.
- tribool(indeterminate\_keyword\_t) :
  - **Objectif** : Constructeur.
  - **Post-condition** : La valeur prise est 'indéterminé'.
  - **Notes** : Nous verrons plus loin quel est ce paramètre 'indéterminé'.
- operator safe\_bool() const :
  - **Objectif** : Opérateur de conversion pour tester la valeur.
  - **Post-condition** : La valeur en retour s'évalue à *true* uniquement si *this* est dans l'état 'vrai'.
  - **Exceptions** : Aucune.
- boost::logic::tribool::value\_t value :
  - **Objectif** : L'état de l'objet à l'aide d'un enum ...
  - **Définition** : enum value\_t { false\_value, true\_value, indeterminate\_value };
  - **Notes** : Bien qu'exposée dans l'interface de la classe, on peut dans la plupart des cas se passer de cette valeur. L'utilisation la plus opportune est dans un switch/case. On notera qu'aucun constructeur ne prend cet enum en paramètre. Tout simplement, car les valeurs 'vrai' et 'faux' sont fixées par le passage d'un *bool* et la valeur 'indéterminé' utilise le type particulier *indeterminate\_keyword\_t* abordé plus bas.



L'état pris par le constructeur par défaut est 'faux' et non 'indéterminé', ce qui n'est pas forcément intuitif.

### VI-C-2 - Type 'indéterminé'

L'identificateur pour l'état 'indéterminé' est :

```
indeterminate
```

On peut donc l'utiliser comme suit :

```
boost::logic::tribool ma_variable(boost::logic::indeterminate);
// ou
ma_variable = boost::logic::indeterminate;
```

Une macro permet de définir son propre mot-clé pour l'état 'indéterminé' :

```
BOOST_TRIBOOL_THIRD_STATE(Name)
```

Ce qui donne :

```
BOOST_TRIBOOL_THIRD_STATE(peut_etre);
int main()
{
    boost::logic::tribool ma_variable(peut_etre);
    ...
}
```

boost::logic::indeterminate ou les mots clés définis avec BOOST\_TRIBOOL\_THIRD\_STATE correspondent aussi à des fonctions permettant de tester si une variable *tribool* est dans l'état 'indéterminé' :

```
BOOST_TRIBOOL_THIRD_STATE(peut_etre)

int main()
{
    boost::logic::tribool ma_variable(peut_etre);
    if(peut_etre(ma_variable)) {
        std::cout<<"peut_etre"<<std::endl;
    }
    if(boost::logic::indeterminate(ma_variable)) {
        std::cout<<"indeterminate"<<std::endl;
    }
    return 0;
}
```

- inline bool indeterminate(tribool x);
  - **Objectif** : Tester si le paramètre est dans l'état 'indéterminé'.
  - **Paramètre** : La valeur à tester.
  - **Post-condition** : La fonction retourne *true* si le paramètre est dans l'état 'indéterminé', *false* sinon.
  - **Exceptions** : Aucune.
  - **Note** : Ceci s'applique aussi aux fonctions définis par BOOST\_TRIBOOL\_THIRD\_STATE.

## VI-C-3 - Opérateur !

tribool operator !(tribool x); :

- **Objectif** : Surcharge de l'opérateur '!'.
  - **Post-condition** :

Etat du paramètre	Etat retourné
'vrai'	'faux'
'faux'	'vrai'
'indéterminé'	'indéterminé'

- **Exceptions** : Aucune.
- **Note** : On remarque que l'état 'indéterminé' est neutre pour l'opérateur.

## VI-C-4 - Opérateur &&

tribool operator &&(tribool x, indeterminate\_keyword\_t);

```
tribool operator &&(tribool x, tribool y);
tribool operator &&(tribool x, bool y);
tribool operator &&(bool x, tribool y);
tribool operator &&(indeterminate_keyword_t, tribool x);
```

- **Objectif** : Surcharge de l'opérateur '&&'.
- **Post-condition** :

Etat opérande gauche	Etat opérande droite	Etat retourné
'vrai'/true	'vrai'/true	'vrai'
'vrai'/true	'faux'/false	'faux'
'vrai'/true	'indéterminé'	'indéterminé'
'faux'/false	'vrai'/true	'faux'
'faux'/false	'faux'/false	'faux'
'faux'/false	'indéterminé'	'faux'
'indéterminé'	'vrai'/true	'indéterminé'
'indéterminé'	'faux'/false	'faux'
'indéterminé'	'indéterminé'	'indéterminé'

- **Exceptions** : Aucune.
- **Note 1** : On remarque que l'état 'faux' est absorbant.
- **Note 2** : Les différentes surcharges permettent l'utilisation de l'opérateur avec un *bool* ou avec un mot-clé de type *indeterminate\_keyword\_t*.
- **Note 3** : Le mot-clé de type *indeterminate\_keyword\_t* ne peut être utilisé que conjointement avec un *tribool*. L'utilisation directe d'un *indeterminate\_keyword\_t* avec un *bool* génère un avertissement.

## VI-C-5 - Opérateur ||

```
tribool operator ||(tribool x, tribool y);
tribool operator ||(tribool x, bool y);
tribool operator ||(bool x, tribool y);
tribool operator ||(indeterminate_keyword_t, tribool x);
tribool operator ||(tribool x, indeterminate_keyword_t);
```

- **Objectif** : Surcharge de l'opérateur '||'.
- **Post-condition** :

Etat opérande gauche	Etat opérande droite	Etat retourné
'vrai'/true	'vrai'/true	'vrai'
'vrai'/true	'faux'/false	'vrai'
'vrai'/true	'indéterminé'	'vrai'
'faux'/false	'vrai'/true	'vrai'
'faux'/false	'faux'/false	'faux'
'faux'/false	'indéterminé'	'indéterminé'
'indéterminé'	'vrai'/true	'vrai'
'indéterminé'	'faux'/false	'indéterminé'
'indéterminé'	'indéterminé'	'indéterminé'

- **Exceptions** : Aucune.
- **Note 1** : On remarque que l'état 'vrai' est absorbant. Les règles des opérateurs sont cohérentes :  $a||b == !(a)&&(!b)$ .
- **Note 2** : Les différentes surcharges permettent l'utilisation de l'opérateur avec un *bool* ou avec un mot-clé de type *indeterminate\_keyword\_t*.


- **Note 3 :** Le mot-clé de type *indeterminate\_keyword\_t* ne peut être utilisé que conjointement avec un *tribool*. L'utilisation directe d'un *indeterminate\_keyword\_t* avec un *bool* génère un avertissement.

## VI-C-6 - Opérateur ==

```
tribool operator ==(tribool x, tribool y);
tribool operator ==(tribool x, bool y);
tribool operator ==(bool x, tribool y);
tribool operator ==(indeterminate_keyword_t, tribool x);
tribool operator ==(tribool x, indeterminate_keyword_t);
```

- **Objectif :** Surcharge de l'opérateur '=='.
- **Post-condition :**

Etat opérande gauche	Etat opérande droite	Etat retourné
'vrai'/true	'vrai'/true	'vrai'
'vrai'/true	'faux'/false	'faux'
'vrai'/true	'indéterminé'	'indéterminé'
'faux'/false	'vrai'/true	'faux'
'faux'/false	'faux'/false	'vrai'
'faux'/false	'indéterminé'	'indéterminé'
'indéterminé'	'vrai'/true	'indéterminé'
'indéterminé'	'faux'/false	'indéterminé'
'indéterminé'	'indéterminé'	'indéterminé'


- **Exceptions :** Aucune.
- **Note 1 :** On remarque que l'état 'indéterminé' est absorbant.  
 **Attention, cela signifie que les tests `if('indéterminé'=='indéterminé')` échouent !**
- **Note 2 :** Les différentes surcharges permettent l'utilisation de l'opérateur avec un *bool* ou avec un mot-clé de type *indeterminate\_keyword\_t*.
- **Note 3 :** Le mot-clé de type *indeterminate\_keyword\_t* ne peut être utilisé que conjointement avec un *tribool*. L'utilisation directe d'un *indeterminate\_keyword\_t* avec un *bool* génère un avertissement.

## VI-C-7 - Opérateur !=

```
tribool operator !=(tribool x, tribool y);
tribool operator !=(tribool x, bool y);
tribool operator !=(bool x, tribool y);
tribool operator !=(indeterminate_keyword_t, tribool x);
tribool operator !=(tribool x, indeterminate_keyword_t);
```

- **Objectif :** Surcharge de l'opérateur '!='.
- **Post-condition :**

Etat opérande gauche	Etat opérande droite	Etat retourné
'vrai'/true	'vrai'/true	'faux'
'vrai'/true	'faux'/false	'vrai'
'vrai'/true	'indéterminé'	'indéterminé'
'faux'/false	'vrai'/true	'vrai'
'faux'/false	'faux'/false	'faux'
'faux'/false	'indéterminé'	'indéterminé'
'indéterminé'	'vrai'/true	'indéterminé'
'indéterminé'	'faux'/false	'indéterminé'
'indéterminé'	'indéterminé'	'indéterminé'

- **Exceptions** : Aucune.
- **Note 1** : On remarque que l'état 'indéterminé' est absorbant.  
 **Attention, cela signifie que les tests `if('indéterminé'!='vrai'/true)`, `if('indéterminé'!='faux'/false)` et `if('indéterminé'!='indéterminé')` échouent !**

La cohérence est assurée : '!=r' prendra la même valeur que '!(l==r)'.

- **Note 2** : Les différentes surcharges permettent l'utilisation de l'opérateur avec un *bool* ou avec un mot-clé de type *indeterminate\_keyword\_t*.
- **Note 3** : Le mot-clé de type *indeterminate\_keyword\_t* ne peut être utilisé que conjointement avec un *tribool*. L'utilisation directe d'un *indeterminate\_keyword\_t* avec un *bool* génère un avertissement.

## VI-D - Les flux et boost::logic::tribool

### VI-D-1 - Les valeurs

Pour rappel, les valeurs *bool* existent sous deux formats dans les flux. Elles peuvent être numériques ou littérales. Soit l'exemple suivant :

Ecriture d'un bool :

```
#include <iostream>

int main()
{
    std::cout<<std::noboolalpha;
    std::cout<<true<<std::endl;
    std::cout<<false<<std::endl;

    std::cout<<std::boolalpha;
    std::cout<<true<<std::endl;
    std::cout<<false<<std::endl;
    return 0;
}
```

Ce programme produit la sortie suivante :

Sortie :

```
Valeur 'true' : 1
Valeur 'false' : 0
Littéral 'true' : true
Littéral 'false' : false
```

De la même façon, une valeur *tribool* peut être utilisée dans les flux soit sous forme numérique, soit sous forme littérale. Lorsque l'état d'une valeur *tribool* correspond à 'vrai' ou à 'faux' alors ce sont les paramètres positionnés pour *bool* qui sont pris en compte. La classe ne gère que l'état 'indéterminé'. La valeur numérique par défaut est '2' et la valeur littérale par défaut est 'indeterminate'. Le code suivant :

### Ecriture d'un tribool :

```
#include <iostream>
#include <boost/logic/tribool.hpp>
#include <boost/logic/tribool_io.hpp>

int main()
{
    boost::logic::tribool ma_variable;
    std::cout<<std::noboolalpha;
    std::cout<<"vrai : "<<boost::logic::tribool(true)<<std::endl;
    std::cout<<"faux : "<<boost::logic::tribool(false)<<std::endl;
    std::cout<<"indéterminé : "<<boost::logic::tribool(boost::logic::indeterminate)<<std::endl;

    std::cout<<std::boolalpha;
    std::cout<<"vrai : "<<boost::logic::tribool(true)<<std::endl;
    std::cout<<"faux : "<<boost::logic::tribool(false)<<std::endl;
    std::cout<<"indéterminé : "<<boost::logic::tribool(boost::logic::indeterminate)<<std::endl;
    return 0;
}
```

produira comme sortie :

### Sortie :

```
vrai : 1
faux : 0
indéterminé : 2
vrai : true
faux : false
indéterminé : indeterminate
```

## VI-D-2 - Entrée

L'opérateur '>>' est surchargé pour le type boost::logic::tribool.

```
template<typename CharT, typename Traits>
inline std::basic_istream<CharT, Traits>&
operator>>(std::basic_istream<CharT, Traits>& in, tribool& x);
```

- **Objectif** : Surcharge de l'opérateur '>>'.
- **Post-condition** : Si la valeur lue est valide, alors la variable est dans l'état correspondant. Sinon, le flag *failbit* est positionné dans le flux. Le flux est retourné.
- **Exceptions** : Dépend de ios::exceptions (voir section sur l'implémentation).

La lecture d'une valeur se fait simplement :

### Lecture d'un tribool :

```
#include <iostream>
#include <sstream>
#include <boost/logic/tribool.hpp>
#include <boost/logic/tribool_io.hpp>

int main()
{
    boost::logic::tribool ma_variable;
    std::stringstream("1")>>std::noboolalpha>>ma_variable; // Etat 'vrai'
    std::stringstream("0")>>std::noboolalpha>>ma_variable; // Etat 'faux'
    std::stringstream("2")>>std::noboolalpha>>ma_variable; // Etat 'indéterminé'

    std::stringstream("true")>>std::boolalpha>>ma_variable; // Etat 'vrai'
    std::stringstream("false")>>std::boolalpha>>ma_variable; // Etat 'faux'
    std::stringstream("indeterminate")>>std::boolalpha>>ma_variable; // Etat 'indéterminé'

    return 0;
}
```

Si le flux n'est pas paramétré avec `std::boolalpha`, alors les valeurs attendues sont 0 pour 'faux', 1 pour 'vrai' et 2 pour 'indéterminé'.

Si le flux est paramétré avec `std::boolalpha` et si l'implémentation de la STL n'implémente pas *locale*, alors les valeurs attendues sont 'false' pour 'faux', 'true' pour 'vrai' et 'indeterminate' pour 'indéterminé'. Ces chaînes sont définies par la bibliothèque Boost.Tribool.

Si le flux est paramétré avec `std::boolalpha` et si l'implémentation de la STL implémente *locale*, alors les valeurs attendues sont celles de la facette `std::num_punct` du flux, soit `std::num_punct<CharT>::falsename()` pour 'faux', `std::num_punct<CharT>::truename()` pour 'vrai'. Pour l'état 'indéterminé', soit une facette spécifique existe (cf. infra.), et elle est utilisée, soit il s'agit de la valeur par défaut 'indeterminate' (définie par la bibliothèque Boost.Tribool).

## VI-D-3 - Sorties

Comme le laisse entendre la section sur les valeurs, l'opérateur '<<' est surchargé pour le type `boost::logic::tribool`.

```
template<typename CharT, typename Traits>
inline std::basic_ostream<CharT, Traits>&
operator<<(std::basic_ostream<CharT, Traits>& out, tribool x)
```

- **Objectif** : Surcharge de l'opérateur '<<'.
- **Post-condition** : Le flux est retourné.
- **Exceptions** : Dépend de `ios::exceptions` (voir section sur l'implémentation).

L'écriture se fait simplement :

### Écriture d'un tribool :

```
#include <iostream>
#include <boost/logic/tribool.hpp>
#include <boost/logic/tribool_io.hpp>

int main()
{
    std::cout<<std::noboolalpha;
    std::cout<<"vrai : "
        <<boost::logic::tribool(true)
    <<std::endl;
    std::cout<<"faux : "
        <<boost::logic::tribool(false)
    <<std::endl;
    std::cout<<"indéterminé : "
        <<boost::logic::tribool(boost::logic::indeterminate)
    <<std::endl;

    std::cout<<std::boolalpha;
    std::cout<<"vrai : "
        <<boost::logic::tribool(true)
    <<std::endl;
    std::cout<<"faux : "
        <<boost::logic::tribool(false)
    <<std::endl;
    std::cout<<"indéterminé : "
        <<boost::logic::tribool(boost::logic::indeterminate)
    <<std::endl;

    return 0;
}
```

La sortie est gérée pour le type du mot-clé 'indéterminé' `indeterminate_keyword_t` :

### Écriture du type 'indéterminé' :

```
#include <boost/logic/tribool.hpp>
#include <boost/logic/tribool_io.hpp>
#include <iostream>
```

#### Ecriture du type 'indéterminé' :

```
BOOST_TRIBOOL_THIRD_STATE(peut_etre)

using boost::logic::operator <<;

int main()
{
    std::cout<<"peut_etre : "<<std::boolalpha
    <<peut_etre
    <<" ( "<<std::noboolalpha<<peut_etre<<" )"
    <<std::endl;
    std::cout<<"peut_etre : "<<std::boolalpha
    <<boost::logic::indeterminate
    <<" ( "<<std::noboolalpha<<boost::logic::indeterminate<<" )"
    <<std::endl;
    return 0;
}
```

L'opérateur << doit être importé dans l'espace de nommage courant, soit par le biais de 'using boost::logic::operator <<;', soit par le biais de 'using namespace boost::logic;'. Autrement, certains compilateurs (Visual C+ 2008 Express Edition, par exemple) prennent une autre surcharge de l'opérateur.

Si l'état est 'vrai' ou 'faux' alors la surcharge de l'opérateur est strictement équivalente à *out << true;* ou respectivement *out << false;*.

Pour l'état 'indéterminé'

Si le flux n'est pas paramétré avec *std::boolalpha*, alors la valeur produite est 2.

Si le flux est paramétré avec *std::boolalpha* et si l'implémentation de la STL n'implémente pas *locale*, alors la valeur produite est 'indeterminate' (définie par la bibliothèque Boost.Tribool).

Si le flux est paramétré avec *std::boolalpha* et si l'implémentation de la STL implémente *locale*, et si la facette spécifique (cf. infra.) est positionnée pour le flux, alors la valeur produite est celle de la facette.

Si le flux est paramétré avec *std::boolalpha* et si l'implémentation de la STL implémente *locale*, et le flux n'a pas de facette spécifique, alors la valeur produite est 'indeterminate' (définie par la bibliothèque Boost.Tribool).

#### VI-D-4 - Facette pour l'état 'indéterminé'

En l'absence de facette ou en l'absence d'implémentation de *locale* par la STL, la fonction suivante est utilisée pour récupérer la valeur littérale de l'état 'indéterminé' :

```
template<typename T> std::basic_string< T > get_default_indeterminate_name();
```

Cette fonction fait partie de l'interface et peut, par conséquent, être utilisée pour récupérer le nom par défaut des développeurs de Boost.

Mais, pour correctement internationaliser un programme, il est préférable, si la STL implémente *locale* d'utiliser une facette. Boost.Tribool définit une classe pour cette facette dont l'interface est :

#### facette tribool pour l'état 'indéterminé' :

```
template<typename CharT>
class indeterminate_name {
public:
    // types
    typedef CharT char_type;
    typedef std::basic_string< CharT > string_type;

    // construct/copy/destruct
    indeterminate_name();
    indeterminate_name(const string_type &);

    // public member functions
    string_type name() const;
    static std::locale::id id;
```



```
facette tribool pour l'état 'indéterminé' :
};
```

Attention, il s'agit de l'interface publique et non de sa réelle implémentation (cf. infra.)

La seule chose à faire est d'instancier une variable de cette classe et de la positionner dans le flux. Deux constructeurs sont possibles. Le constructeur par défaut renseignera la valeur avec la fonction précédente `get_default_indeterminate_name`. Le second constructeur permet de spécifier sa propre chaîne.

L'exemple ci-dessous, sous réserve que la STL implémente *locale*, permet de positionner "peut-être" pour la valeur 'indéterminé' :

#### Utilisation d'une facette :

```
#include <boost/logic/tribool.hpp>
#include <boost/logic/tribool_io.hpp>
#include <iostream>
#include <sstream>

BOOST_TRIBOOL_THIRD_STATE(peut_etre)

using boost::logic::operator <<;

int main()
{
    // spécialisation de locale avec notre facette :
    // la facette est détruite lorsque mon_locale est détruite.
    std::locale mon_locale(std::locale(), new boost::logic::indeterminate_name<char>("peut-être"));
    // positionnement de nos paramètres locaux dans notre flux :
    std::cout.imbue(mon_locale);
    // sortie :
    std::cout<<"peut_etre : "<<std::boolalpha<<boost::logic::tribool(peut_etre)<<std::endl;
    std::cout<<"peut_etre : "<<std::boolalpha<<peut_etre<<std::endl;

    boost::logic::tribool ma_variable;
    std::stringstream mon_flux("peut-être");
    mon_flux.imbue(mon_locale);
    mon_flux>>std::boolalpha>>ma_variable; // Etat 'indéterminé'

    return 0;
}
```


## VI-D-5 - STL sans implémentation de locale


Si la STL n'implémente pas *locale*, Boost.Tribool doit être utilisé avec la directive de compilation `BOOST_NO_STD_LOCALE`.

## VI-E - En résumé

En-tête de la classe :	boost/logic/tribool.hpp
En-tête i/o :	boost/logic/tribool_io.hpp
Espace de nommage :	boost::logic boost
Définition d'une variable :	boost::logic::tribool ma_variable;
Mot-clé pour 'indéterminé' :	boost::logic::indeterminate
Définition de son propre mot-clé :	BOOST_TRIBOOL_THIRD_STATE(Name)
Tester l'état 'indéterminé' :	if(boost::logic::indeterminate(ma_variable))
Valeur par défaut dans un flux :	Etat 'faux' >> 'false' >> 0 Etat 'vrai' >> 'true' >> 1 Etat 'indéterminé' >> 'indeterminate' >> 2

 **Le constructeur par défaut positionne l'état à 'faux' !**

 **L'état 'indéterminé' est absorbant pour l'opérateur '==' :**  
`if(boost::logic::tribool(boost::logic::indeterminate)==boost::logic::tribool(boost::logic::indeterminate))`  
 échoue !

 **L'état 'indéterminé' est absorbant pour l'opérateur '!=' :**  
`if(boost::logic::tribool(boost::logic::indeterminate)!=boost::logic::tribool(true))`  
 échoue !  
`if(boost::logic::tribool(boost::logic::indeterminate)!=boost::logic::tribool(false))`  
 échoue !  
`if(boost::logic::tribool(boost::logic::indeterminate)`  
`=boost::logic::tribool(boost::logic::indeterminate)) échoue !`

## VII - Comment ça marche ?

### VII-A - Les fichiers

- boost/logic/tribool.hpp : définition de la classe boost::logic::tribool, du type boost::logic::indeterminate et des surcharges des opérateurs ;
- boost/logic/tribool\_fwd.hpp : déclaration de la classe ;
- boost/logic/tribool\_io.hpp : définition des surcharges nécessaires aux opérations d'entrée/sortie avec les flux.

Boost.Tribool se compose essentiellement de trois fichiers :

### VII-B - La classe

Voici la définition de la classe :

```
Classe tribool :
class tribool
{
private:
    /// INTERNAL ONLY
    struct dummy {
        void nonnull() {};
    };

    typedef void (dummy::*safe_bool)();

public:
    tribool() : value(false_value) {}
    tribool(bool value) : value(value? true_value : false_value) {}
    tribool(indeterminate_keyword_t) : value(indeterminate_value) {}
    operator safe_bool() const
    {
        return value == true_value? &dummy::nonnull : 0;
    }
    enum value_t { false_value, true_value, indeterminate_value } value;
};
```

Rien de bien complexe dans cette classe. La seule subtilité est la définition de `safe_bool`. Cela permet de le rendre interne à la classe et privé en évitant ainsi d'éventuel conflit ou de mésusage.

## VII-C - Le type 'indéterminé'

Une structure interne est d'abord déclarée :

Structure interne :

```
struct indeterminate_t
{
    #if BOOST_WORKAROUND(__BORLANDC__, < 0x0600)
        char dummy_; // BCB would use 8 bytes by default
    #endif
};
```

Elle n'a pas pour but d'être utilisée ! Elle sert à définir le type d'un mot-clé pour l'état 'indéterminé' :

Type d'un mot-clé pour l'état 'indéterminé' :

```
typedef bool (*indeterminate_keyword_t)(tribool, detail::indeterminate_t);
```

C'est un pointeur de fonction pour pouvoir utiliser, comme nous l'avons vu, le mot-clé aussi bien comme valeur que comme fonction :

Rappel :

```
tribool ma_variable(indeterminate); // utilisation en valeur
if(indeterminate(ma_variable)) { // utilisation en fonction
    ...
}
```

L'utilisation du type interne detail::indeterminate\_t permet d'éviter la définition par l'utilisateur d'une fonction qui aurait la même signature.

Le mot-clé par défaut 'indeterminate' est tout simplement une fonction suivant la définition précédente :

Mot-clé 'indeterminate' :

```
inline bool
indeterminate(tribool x,
              detail::indeterminate_t dummy = detail::indeterminate_t());

inline bool indeterminate(tribool x, detail::indeterminate_t)
{
    return x.value == tribool::indeterminate_value;
}
```

Enfin, la macro suivante définit un mot-clé utilisateur en déclarant une fonction du nom voulu suivant la signature *indeterminate\_keyword\_t* :

Macro de définition d'un mot-clé utilisateur :

```
#define BOOST_TRIBOOL_THIRD_STATE(Name) \
inline bool \
Name(boost::logic::tribool x, \
      boost::logic::detail::indeterminate_t dummy = \
      boost::logic::detail::indeterminate_t()) \
{ return x.value == boost::logic::tribool::indeterminate_value; }
```

## VII-D - Les opérateurs.

La surcharge des opérateurs est assez simple et ne présente aucune particularité :

Surcharge des opérateurs :

```
inline tribool operator!(tribool x)
{
    return x.value == tribool::false_value? tribool(true)
}
```

### Surcharge des opérateurs :

```

        :x.value == tribool::true_value? tribool(false)
        :tribool(indeterminate);
    }

inline tribool operator&&(tribool x, tribool y)
{
    if (static_cast<bool>(!x) || static_cast<bool>(!y))
        return false;
    else if (static_cast<bool>(x) && static_cast<bool>(y))
        return true;
    else
        return indeterminate;
}

inline tribool operator&&(tribool x, bool y)
{ return y? x : tribool(false); }

inline tribool operator&&(bool x, tribool y)
{ return x? y : tribool(false); }

inline tribool operator&&(indeterminate_keyword_t, tribool x)
{ return !x? tribool(false) : tribool(indeterminate); }

inline tribool operator&&(tribool x, indeterminate_keyword_t)
{ return !x? tribool(false) : tribool(indeterminate); }

inline tribool operator|| (tribool x, tribool y)
{
    if (static_cast<bool>(!x) && static_cast<bool>(!y))
        return false;
    else if (static_cast<bool>(x) || static_cast<bool>(y))
        return true;
    else
        return indeterminate;
}

inline tribool operator|| (tribool x, bool y)
{ return y? tribool(true) : x; }

inline tribool operator|| (bool x, tribool y)
{ return x? tribool(true) : y; }

inline tribool operator|| (indeterminate_keyword_t, tribool x)
{ return x? tribool(true) : tribool(indeterminate); }

inline tribool operator|| (tribool x, indeterminate_keyword_t)
{ return x? tribool(true) : tribool(indeterminate); }

inline tribool operator==(tribool x, tribool y)
{
    if (indeterminate(x) || indeterminate(y))
        return indeterminate;
    else
        return (x && y) || (!x && !y);
}

inline tribool operator==(tribool x, bool y) { return x == tribool(y); }

inline tribool operator==(bool x, tribool y) { return tribool(x) == y; }

inline tribool operator==(indeterminate_keyword_t, tribool x)
{ return tribool(indeterminate) == x; }

inline tribool operator==(tribool x, indeterminate_keyword_t)
{ return tribool(indeterminate) == x; }

inline tribool operator!=(tribool x, tribool y)
{
    if (indeterminate(x) || indeterminate(y))
        return indeterminate;
}

```

### Surcharge des opérateurs :

```

else
    return !((x && y) || (!x && !y));
}

inline tribool operator!=(tribool x, bool y) { return x != tribool(y); }

inline tribool operator!=(bool x, tribool y) { return tribool(x) != y; }

inline tribool operator!=(indeterminate_keyword_t, tribool x)
{ return tribool(indeterminate) != x; }

inline tribool operator!=(tribool x, indeterminate_keyword_t)
{ return x != tribool(indeterminate); }

```

## VII-E - Les flux

### VII-E-1 - Les valeurs littérales

Si la STL n'implémente pas *locale*, alors la bibliothèque doit être utilisée avec la directive de compilation `BOOST_NO_STD_LOCALE`. Les fonctions suivantes sont alors définies pour obtenir les valeurs littérales :

### Fonctions d'obtention des valeurs littérales :

```

#ifdef BOOST_NO_STD_LOCALE
template<typename T> std::basic_string<T> default_false_name();

template<>
inline std::basic_string<char> default_false_name<char>()
{ return "false"; }

# ifndef BOOST_NO_WCHAR_T
template<>
inline std::basic_string<wchar_t> default_false_name<wchar_t>()
{ return L"false"; }
# endif

template<typename T> std::basic_string<T> default_true_name();

template<>
inline std::basic_string<char> default_true_name<char>()
{ return "true"; }

# ifndef BOOST_NO_WCHAR_T
template<>
inline std::basic_string<wchar_t> default_true_name<wchar_t>()
{ return L"true"; }
#endif

```

Par défaut, les fonctions sont spécialisées sur *char* et *wchar\_t*.

Concernant l'état 'indéterminé', la valeur littérale doit, en tout état de cause, être produite car la facette spécifique peut ne pas avoir été positionnée dans le flux. Les fonctions suivantes sont donc systématiquement définies (indépendamment de la directive `BOOST_NO_STD_LOCALE`) :

### Fonction d'obtention de la valeur littérale pour l'état 'indéterminé' :

```

template<typename T> std::basic_string<T> get_default_indeterminate_name();
template<>
inline std::basic_string<char> get_default_indeterminate_name<char>()
{ return "indeterminate"; }
#ifdef BOOST_WORKAROUND(BOOST_MSVC, < 1300)
// VC++ 6.0 chokes on the specialization below, so we're stuck without
// wchar_t support. What a pain. TODO: it might just need a the template
// parameter as function parameter...
#else

```

#### Fonction d'obtention de la valeur littérale pour l'état 'indéterminé' :

```
# ifndef BOOST_NO_WCHAR_T
template<>
inline std::basic_string<wchar_t> get_default_indeterminate_name<wchar_t>()
{ return L"indeterminate"; }
# endif
#endif
```

Les fonctions sont spécialisées par défaut pour *char* et *wchar\_t*, sous réserve d'un compilateur différents de Visual C++ 6.0.

## VII-E-2 - La facette

La classe suivante définit la facette utilisée ensuite dans les opérateurs d'entrée et de sortie :

#### Facette pour l'état 'indéterminé' :

```
template<typename CharT>
class indeterminate_name : public std::locale::facet, private boost::noncopyable
{
public:
    typedef CharT char_type;
    typedef std::basic_string<CharT> string_type;

    /// Construct the facet with the default name
    indeterminate_name() : name_(get_default_indeterminate_name<CharT>()) {}


    /// Construct the facet with the given name for the indeterminate value
    explicit indeterminate_name(const string_type& name) : name_(name) {}


    /// Returns the name for the indeterminate value
    string_type name() const { return name_; }

    /// Uniquely identifies this facet with the locale.
    static std::locale::id id;

private:
    string_type name_;
};

template<typename CharT> std::locale::id indeterminate_name<CharT>::id;
```

La classe dérive de *std::locale::facet*, comme il se doit pour définir sa propre facette. Des informations sur ces aspects sont disponibles dans le tutoriel de developpez.com suivant :  **Tutoriel locale**.

Elle dérive aussi de *boost::noncopyable*. Son nom assez explicite reflète l'objectif de cette classe : ne pas permettre à une classe qui en dérive d'être recopiée. Le constructeur par copie et l'opérateur d'affectation sont rendus privés. Cette classe fait partie de la bibliothèque Boost.Utility. Des informations supplémentaires sont disponibles sur le site de  **Boost.Class noncopyable**.

L'attribut *name\_* contient la valeur littérale effective de la facette. Avec le constructeur par défaut, la valeur est récupérée avec la fonction *get\_default\_indeterminate\_name* vu ci-dessus. Le second constructeur permet de positionner sa propre valeur littérale.

L'accessor *string\_type name() const* permet de récupérer cet attribut.

L'attribut *id* concerne la gestion de la facette par *locale*.


## VII-E-3 - L'opérateur de sortie

L'opérateur '<<' est surchargé comme suit :

### Redéfinition de l'opérateur de sortie :

```
template<typename CharT, typename Traits>
inline std::basic_ostream<CharT, Traits>&
operator<<(std::basic_ostream<CharT, Traits>& out, tribool x)
{
    if (!indeterminate(x)) {
        out << static_cast<bool>(x);
    } else {
        typename std::basic_ostream<CharT, Traits>::sentry cerberus(out);
        if (cerberus) {
            if (out.flags() & std::ios_base::boolalpha) {
#ifdef BOOST_NO_STD_LOCALE
                if (BOOST_HAS_FACET(indeterminate_name<CharT>, out.getloc())) {
                    const indeterminate_name<CharT>& facet =
                        BOOST_USE_FACET(indeterminate_name<CharT>, out.getloc());
                    out << facet.name();
                } else {
                    out << get_default_indeterminate_name<CharT>();
                }
            }
            #else
                out << get_default_indeterminate_name<CharT>();
            #endif
        }
        else
            out << 2;
    }
    return out;
}
```

La première ligne de la fonction montre que quelque soit l'implémentation de la STL, la sortie pour un état 'vrai' ou 'faux' est strictement identique à celle d'un `bool`.

La classe `std::basic_ostream<CharT, Traits>::sentry` doit être utilisée systématiquement lorsque l'opérateur '<<' est surchargé. Ce point peut être approfondi dans le cours suivant de developpez.com :  [les flux d'entrée/sortie](#)

L'écriture du littéral pour une variable `tribool` dans l'état 'indéterminé' suit ce qui a été précédemment expliqué. Le littéral de la facette est récupérée dès lors que la STL implémente *locale* et que la facette existe dans le flux. Sinon, la fonction `get_default_indeterminate_name` retourne ce littéral.

L'opérateur est aussi surchargé pour la valeur 'indéterminé' :

### Surcharge de l'opérateur de sortie pour la valeur 'indéterminé' :

```
template<typename CharT, typename Traits>
inline std::basic_ostream<CharT, Traits>&
operator<<(std::basic_ostream<CharT, Traits>& out,
           bool (*) (tribool, detail::indeterminate_t))
{ return out << tribool(indeterminate); }
```

Une instance de `tribool` permet de ne pas avoir à recopier de code ...

## VII-E-4 - L'opérateur d'entrée

L'opérateur '>>' est surchargé comme suit :

### Surcharge de l'opérateur d'entrée :

```
template<typename CharT, typename Traits>
inline std::basic_istream<CharT, Traits>&
operator>>(std::basic_istream<CharT, Traits>& in, tribool& x)
{
    if (in.flags() & std::ios_base::boolalpha) {
        typename std::basic_istream<CharT, Traits>::sentry cerberus(in);
        if (cerberus) {
            typedef std::basic_string<CharT> string_type;

```

## Surcharge de l'opérateur d'entrée :

```
#ifndef BOOST_NO_STD_LOCALE
    const std::num_punct<CharT>& num_punct_facet =
        BOOST_USE_FACET(std::num_punct<CharT>, in.getloc());

    string_type falsename = num_punct_facet.falsename();
    string_type truename = num_punct_facet.truename();

    string_type othername;
    if (BOOST_HAS_FACET(indeterminate_name<CharT>, in.getloc())) {
        othername =
            BOOST_USE_FACET(indeterminate_name<CharT>, in.getloc()).name();
    } else {
        othername = get_default_indeterminate_name<CharT>();
    }
#else
    string_type falsename = default_false_name<CharT>();
    string_type truename = default_true_name<CharT>();
    string_type othername = get_default_indeterminate_name<CharT>();
#endif

    typename string_type::size_type pos = 0;
    bool falsename_ok = true, truename_ok = true, othername_ok = true;

    // Modeled after the code from Library DR 17
    while (falsename_ok && pos < falsename.size()
        || truename_ok && pos < truename.size()
        || othername_ok && pos < othername.size()) {
        typename Traits::int_type c = in.get();
        if (c == Traits::eof())
            return in;

        bool matched = false;
        if (falsename_ok && pos < falsename.size()) {
            if (Traits::eq(Traits::to_char_type(c), falsename[pos]))
                matched = true;
            else
                falsename_ok = false;
        }

        if (truename_ok && pos < truename.size()) {
            if (Traits::eq(Traits::to_char_type(c), truename[pos]))
                matched = true;
            else
                truename_ok = false;
        }

        if (othername_ok && pos < othername.size()) {
            if (Traits::eq(Traits::to_char_type(c), othername[pos]))
                matched = true;
            else
                othername_ok = false;
        }

        if (matched) { ++pos; }
        if (pos > falsename.size()) falsename_ok = false;
        if (pos > truename.size()) truename_ok = false;
        if (pos > othername.size()) othername_ok = false;
    }

    if (pos == 0)
        in.setstate(std::ios_base::failbit);
    else {
        if (falsename_ok) x = false;
        else if (truename_ok) x = true;
        else if (othername_ok) x = indeterminate;
        else in.setstate(std::ios_base::failbit);
    }
} else {
```



### Surcharge de l'opérateur d'entrée :

```
long value;
if (in >> value) {
    switch (value) {
        case 0: x = false; break;
        case 1: x = true; break;
        case 2: x = indeterminate; break;
        default: in.setstate(std::ios_base::failbit); break;
    }
}

return in;
}
```

Commençons par le plus simple : le drapeau `std::ios_base::boolalpha` n'est pas levé :

```
long value;
if (in >> value) {
    switch (value) {
        case 0: x = false; break;
        case 1: x = true; break;
        case 2: x = indeterminate; break;
        default: in.setstate(std::ios_base::failbit); break;
    }
}
```

Le paramètre *tribool* prend la valeur 'faux' pour une valeur numérique de 0, 'vrai' pour 1 et 'indéterminé' pour 2. Si l'entier lu est d'une valeur différente de ces dernières, le drapeau `std::ios_base::failbit` est levé.

Lorsque la lecture se fait à partir du littéral, le drapeau `std::ios_base::boolalpha` est levé. La fonction commence par récupérer les valeurs à reconnaître. Lorsque la STL n'implémente pas *locale*, les fonctions sont utilisées pour récupérer les chaînes :

```
string_type falsename = default_false_name<CharT>();
string_type truename = default_true_name<CharT>();
string_type othername = get_default_indeterminate_name<CharT>();
```

Lorsque la STL implémente *locale*, les littéraux pour l'état 'vrai' et 'faux' sont repris de la facette les définissant pour le type *bool* :

```
const std::num_punct<CharT>& num_punct_facet =
    BOOST_USE_FACET<std::num_punct<CharT>, in.getloc()>();

string_type falsename = num_punct_facet.falsename();
string_type truename = num_punct_facet.truename();
```

Le littéral pour l'état 'indéterminé' est récupéré par la facette *indeterminate\_name* si elle est définie, sinon par *get\_default\_indeterminate\_name* :

```
string_type othername;
if (BOOST_HAS_FACET(indeterminate_name<CharT>, in.getloc())) {
    othername =
        BOOST_USE_FACET(indeterminate_name<CharT>, in.getloc()).name();
} else {
    othername = get_default_indeterminate_name<CharT>();
}
```

Ensuite, il s'agit d'un simple automate de reconnaissance des trois valeurs possibles :

```
typename string_type::size_type pos = 0;
bool falsename_ok = true, truename_ok = true, othername_ok = true;
```

```
// Modeled after the code from Library DR 17
while (falsename_ok && pos < falsename.size()
      || truename_ok && pos < truename.size()
      || othername_ok && pos < othername.size()) {
    typename Traits::int_type c = in.get();
    if (c == Traits::eof())
        return in;

    bool matched = false;
    if (falsename_ok && pos < falsename.size()) {
        if (Traits::eq(Traits::to_char_type(c), falsename[pos]))
            matched = true;
        else
            falsename_ok = false;
    }

    if (truename_ok && pos < truename.size()) {
        if (Traits::eq(Traits::to_char_type(c), truename[pos]))
            matched = true;
        else
            truename_ok = false;
    }

    if (othername_ok && pos < othername.size()) {
        if (Traits::eq(Traits::to_char_type(c), othername[pos]))
            matched = true;
        else
            othername_ok = false;
    }

    if (matched) { ++pos; }
    if (pos > falsename.size()) falsename_ok = false;
    if (pos > truename.size()) truename_ok = false;
    if (pos > othername.size()) othername_ok = false;
}
```

Finalement, le paramètre *tribool* est renseigné si la valeur a été correctement lue, sinon le `std::ios_base::failbit` drapeau est levé :

```
if (pos == 0)
    in.setstate(std::ios_base::failbit);
else {
    if (falsename_ok) x = false;
    else if (truename_ok) x = true;
    else if (othername_ok) x = indeterminate;
    else in.setstate(std::ios_base::failbit);
}
```

## VIII - Soyons logique !

Essayons de comprendre les choix opérés sur les définitions des différents opérateurs: &&, ||, !, == et !=.

### VIII-A - Rappel sur l'algèbre de Boole

Pour ne pas effrayer les mathématiciens, disons tout de suite, que dans le cadre de ce tutoriel, l'algèbre de Boole s'entend plus sur le calcul booléen tel qu'il peut être présenté dans les cours d'informatique ou d'électronique. Il ne part pas des structures algébriques telles que définies en mathématiques ce qui serait plus exact et plus complet mais risquerait par la technicité de l'exposé de masquer l'objectif premier de cette partie : expliquer les choix opérés pour construire les opérateurs.

On part de l'ensemble  $B = \{V, F\}$ .

Par **définition**, on construit les 3 fonctions suivantes :

- ET :  $B \times B \rightarrow B$

	V	F
V	V	F
F	F	F

Remarques :

F est un élément absorbant pour ET : Quelque soit x de B,  $ET(x, F) = F$ .

V est un élément neutre pour ET : Quelque soit x de B,  $ET(x, V) = x$ .

ET est commutatif :  $ET(x, y) = ET(y, x)$ .

ET est idempotente :  $ET(x, x) = x$ .

- OU :  $B \times B \rightarrow B$

	V	F
V	V	V
F	V	F

Remarques :

V est un élément absorbant pour OU : Quelque soit x de B,  $OU(x, V) = V$ .

F est un élément neutre pour OU : Quelque soit x de B,  $OU(x, F) = x$ .

OU est commutatif :  $OU(x, y) = OU(y, x)$ .

OU est idempotente :  $OU(x, x) = x$ .

- NON :  $B \rightarrow B$

$NON(V) = F$

$NON(F) = V$

Remarque :  $NON(NON(x)) = x$ .

L'opérateur  $==$  se construit par définition à partir des trois fonctions précédentes :

$x == y : B \times B \rightarrow B$

$x == y = ((x \text{ ET } y) \text{ OU } (NON(x) \text{ ET } NON(y)))$

De la même façon, on construit l'opérateur  $!=$  par définition :

$x != y : B \times B \rightarrow B$

$x == y = NON(x == y) = NON(((x \text{ ET } y) \text{ OU } (NON(x) \text{ ET } NON(y)))) = (x \text{ OU } y) \text{ ET } NON(x \text{ ET } y)$ .

Le lecteur avisé reconnaît ici la définition habituelle de XOR.

En appliquant les tables de vérités, on s'aperçoit que nos définitions ont bien le sens commun qu'on attribue aux opérateurs.

## VIII-B - L'algèbre tri-booléenne

On définit notre ensemble de départ  $T = \{V, F, I\}$

Pour définir nos tables de vérités, on va chercher à maintenir la cohérence des propriétés citées ci-dessus. On va donc reprendre les différentes propriétés des lois de composition pour remplir nos trous.

### VIII-B-1 - L'opérateur ET

Pour commencer, remplissons notre table de vérité pour l'opérateur ET avec celle présentée ci-dessus pour les éléments V et F :

	V	F	I
V	V	F	
F	F	F	
I			

Pour remplir les éléments manquants, on part des remarques présentées pour l'opérateur ET :

- ET est commutatif :  $ET(l,y) = ET(y,l)$
- F est un élément absorbant pour ET :  $ET(F,l) = ET(l,F) = F$
- V est un élément neutre pour ET :  $ET(V,l) = ET(l,V) = l$
- ET est idempotente :  $ET(l,l) = l$ .

On obtient :

	V	F	I
V	V	F	I
F	F	F	F
I	I	F	I

Voilà notre définition de l'opérateur && pleinement spécifiée.

## VIII-B-2 - L'opérateur OU

On opère de la même façon pour OU en partant de la table de vérité des booléens :

	V	F	I
V	V	V	
F	V	F	
I			

Pour remplir les éléments manquants, on part des remarques présentées pour l'opérateur OU :

- OU est commutatif :  $OU(l,y) = OU(y,l)$
- V est un élément absorbant pour OU :  $OU(V,l) = OU(l,V) = V$
- F est un élément neutre pour OU :  $OU(F,l) = OU(l,F) = l$
- OU est idempotente :  $OU(l,l) = l$ .

On obtient :

	V	F	I
V	V	V	V
F	V	F	I
I	V	I	I

Grace à cette table, l'opérateur || est totalement spécifié.

## VIII-B-3 - L'opérateur NON

Partons de la propriété  $NON(NON(x))=x$  et raisonnons par l'absurde

- 1 Soit  $NON(I) = V$   
Il s'en suit que  $NON(NON(I)) = NON(V) = F$  !  
*On ne respecte pas notre propriété  $NON(NON(x))=x$ .*
- 2 Soit  $NON(I) = F$   
Il s'en suit que  $NON(NON(I)) = NON(F) = V$  !  
*On ne respecte pas notre propriété  $NON(NON(x))=x$ .*
- 3 Soit  $NON(I) = I$   
Il s'en suit que  $NON(NON(I)) = NON(I) = I$  !

On respecte notre propriété  $NON(NON(x))=x$ .

Notre définition de NON devient :

- $NON : B \rightarrow B$
- $NON(V) = F$
- $NON(F) = V$
- $NON(I) = I$

L'opérateur ! est maintenant complètement spécifié.

#### VIII-B-4 - L'opérateur ==

On reprend la définition donnée pour le calcul booléen de l'opérateur == :

$x==y : B \times B \rightarrow B$

$x==y = ((x \text{ ET } y) \text{ OU } (NON(x) \text{ ET } NON(y)))$

On applique les tables de vérités ci-dessus et on obtient :

	V	F	I
V	V	F	I
F	F	V	I
I	I	I	I

Cette table de vérité explique le choix de la surcharge de l'opérateur == pour la classe Boost.Tribool. On comprend maintenant pourquoi `boost::logic::tribool(indeterminate) == boost::logic::tribool(indeterminate)` vaut *indeterminate*, ce qui nous avait surpris en début de ce tutoriel !

#### VIII-B-5 - L'opérateur !=

Comme précédemment, on repart de la définition de != ( $NON(==)$ ) et on applique les tables de vérités

	V	F	I
V	F	V	I
F	V	F	I
I	I	I	I