

Tutoriel Boost.Optional

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

Date de publication : 22 décembre 2008

Dernière mise à jour : 17 février 2009

Cet article présente la bibliothèque Boost.Optional 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.....	4
IV-C - Valeur invalide : une indétermination !.....	4
V - Premiers pas.....	5
V-A - Une variable boost::optional.....	5
V-B - Initialisation d'une variable boost::optional.....	5
V-C - Affecter une valeur.....	6
V-D - Vérifier l'état d'une variable.....	6
V-E - Récupérer la valeur.....	7
V-F - Comparer deux variables boost::optional.....	8
V-G - Utilisation dans un conteneur.....	8
VI - Quelques exemples.....	9
VI-A - Valeur en retour indéterminée.....	9
VI-B - Variable locale non initialisée.....	10
VI-C - Membre d'une classe optionnel.....	10
VI-D - Réassigner une référence.....	11
VII - Description de la bibliothèque.....	11
VII-A - En-tête.....	11
VII-B - Espace de nommage.....	11
VII-C - Interface.....	12
VII-C-1 - La classe.....	12
VII-C-2 - Convention.....	12
VII-C-3 - Les constructeurs.....	13
VII-C-4 - Surcharges de l'opérateur '='.....	15
VII-C-5 - Les accesseurs.....	17
VII-C-5-a - Par valeur.....	17
VII-C-5-b - Par adresse.....	18
VII-C-6 - Vérification de l'état.....	19
VII-C-7 - Méthodes obsolètes.....	20
VII-C-8 - Accesseurs externes.....	20
VII-C-9 - Aide à la construction.....	21
VII-C-10 - Opérateurs relationnels.....	21
VII-C-11 - Fonction d'échange : swap.....	23
VII-D - Interaction avec les flux.....	23
VII-E - Limitations et contraintes.....	23
VII-F - En résumé.....	24
VIII - Comment ça marche ?.....	24
VIII-A - Zone de stockage de la valeur.....	25
VIII-B - Gérer les références.....	27
VIII-C - Vérifier l'état.....	29
VIII-D - Opérateur d'affectation.....	30
VIII-D-1 - Déterminer la bonne surcharge.....	30
VIII-D-2 - Réassigner une référence.....	31

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.Optional**.

III - Remerciements

Je remercie **Alp** pour ses encouragements, **ram-0000** pour sa relecture, **_Mac_** pour la grand-mère et l'aurteaugraffe 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. Plus particulièrement, je remercie **JolyLoic** qui m'a dirigé vers l'article  **The Safe Bool Idiom**, de Bjorn Karlsson.

IV - Introduction

IV-A - Pré-requis

Avant toute chose, à l'instar de l'ensemble des bibliothèques Boost, le lecteur doit déjà être familier avec le langage C++ et plus particulièrement se sentir à l'aise avec les templates. La bibliothèque Boost.Optional utilise les bases de la programmation template. Il ne faut pas que celles-ci puissent être un obstacle à la compréhension.

L'utilisation de la bibliothèque Boost.Optional ne nécessite pas de connaissances particulières sur les autres bibliothèques Boost. Cependant, Boost.Optional s'interface avec la bibliothèque Boost.In Place Factory. Le tutoriel suivant présente cette bibliothèque :  **Tutoriel Boost.In Place Factory**

L'implémentation, quant à elle, s'appuie sur les bibliothèques suivantes :

- Boost.MPL
- Boost.Type Traits
- Boost.Utility.Assert

Sa compréhension nécessite donc d'avoir déjà côtoyé ces dernières.

L'implémentation utilise le concept de classes de traits et politiques ainsi que le concept de méta-programmation. Les tutoriels suivants présentent ces concepts :

-  **Présentation des classes de Traits et de Politiques en C++ ;**

-  **La méta-programmation en C++.**

IV-B - Objectifs

Deux idées maîtresses justifient Boost.Optional.

La première est liée au besoin d'avoir des variables à signification particulière représentant l'idée de valeur invalide. En formalisant le concept de valeur indéterminée, Boost.Optional permet de gérer les valeurs invalides sans avoir à étendre le champ des valeurs que peut prendre une donnée.

La seconde motivation tourne autour de la notion de variable non initialisée. La norme dit que l'utilisation d'une variable non initialisée est indéterminée. Or ce concept est en soi absent du langage. On ne peut pas faire de test pour savoir si une variable a été initialisée ou non. Boost.Optional formalise le concept de variable non initialisée et permet de gérer cette indétermination.

IV-C - Valeur invalide : une indétermination !

Considérons la fonction suivante (tirée de la documentation officielle) :

```
unsigned char get_async_input();
```

Cette fonction retourne l'octet disponible sur une interface, en mode asynchrone. Que faire si aucune donnée n'est disponible sur l'interface ? Plusieurs solutions sont couramment implémentées :

- l'utilisation d'une valeur spécifique du champ des valeurs possibles indiquant une valeur invalide :

```
static const unsigned char NO_DATA = (unsigned char)-1;
```

- l'extension du champ des valeurs possibles, les valeurs hors champ initial signifient l'invalidité :

```
unsigned int get_async_input();  
// Une valeur plus grande que 255 indique une valeur invalide.
```

- adjoindre un indicateur lié à l'état de validité de la valeur :

```
std::pair<bool, unsigned char> get_async_input();
```

Ces solutions souffrent d'inconvénients. La première solution n'est pas toujours possible : si toutes les valeurs du type sont significatives, ou si la fonction retourne un objet plus complexe dont on ne peut construire aisément une instance invalide. La seconde élargit le champ des valeurs en retour et rend moins immédiate la compréhension de ce résultat : difficile de se rendre compte que les seules valeurs valides sont les octets. La troisième apparaît un peu lourde. Enfin, chacune de ces méthodes n'empêche pas l'utilisateur de traiter la valeur en retour comme une valeur valide. L'erreur peut passer inaperçue assez longtemps et devenir très difficile à diagnostiquer !

La norme précise que l'utilisation d'une variable non initialisée est indéterminée. C'est une notion très intéressante pour notre exemple. Avec les solutions proposées, la valeur en retour est initialisée, donc déterminée, mais elle revêt un caractère invalide par sa sémantique. En rendant notre valeur en retour non initialisée, de facto son utilisation est indéterminée, ce qui devient une erreur. La puissance conceptuelle de Boost.Optional est d'avoir marié cette notion d'indétermination avec celle de valeur invalide que peut prendre une variable ! Boost.Optional formalise la notion de variable non initialisée. Donc, avec Boost.Optional, la solution consiste à retourner une valeur invalide. Bien sûr, l'interface permet de diagnostiquer l'état de la variable pour une utilisation adéquate.

Une dernière parenthèse - toujours dans le fil de la documentation officielle - pour limiter notre champ d'application: le fait qu'une valeur ne puisse être déterminée peut relever de l'erreur et doit être géré en tant que tel. Ainsi, `sqrt(-1)` est hors de notre propos car l'indétermination de la valeur en retour est directement liée au non respect de la pré-condition de la fonction : le paramètre n'est pas valide. Ce type d'erreur est habituellement géré selon les trois solutions suivantes :

- en disant que la valeur en retour de la fonction est indéterminée pour un paramètre incorrect (solution documentaire) ;
- par le biais d'une assertion en mode debug (diagnostique par test) ;
- par le biais d'une exception pour permettre au développeur de gérer cette erreur (gestion dans le programme).

Il est important de comprendre que les cas d'erreurs à l'origine d'une indétermination doivent être maintenus en tant que tels car ils restent conceptuellement la meilleure solution acceptable.

V - Premiers pas

V-A - Une variable boost::optional

La déclaration d'une variable se fait tout simplement comme suit :

```
boost::optional<Type> variable;
```

Cela donne par exemple :

```
boost::optional<int> opt_int;  
boost::optional<std::string> opt_string;  
boost::optional<CMaClasse> opt_ma_classe;  
boost::optional<CMaClasse&> opt_ref_ma_classe;
```

V-B - Initialisation d'une variable boost::optional

La variable peut être initialisée à la construction :

```
boost::optional<int> opt_int(3); // opt_int est initialisée à la valeur '3'  
CMaClasse valeur(/*... paramètres du constructeur ... */);  
boost::optional<CMaClasse> opt_ma_classe(valeur);  
    //opt_ma_classe est initialisée avec une copie de 'valeur'
```

Boost.In Place factory permet l'initialisation avec un constructeur adéquat :

```
class CMaClasse  
{  
public:  
    CMaClasse(int vari_, std::string varstr_);  
    // ... définition de CMaClasse  
};  
boost::optional<CMaClasse> opt_ma_classe(boost::in_place(1,"chaîne"));  
    // Le constructeur CMaClasse(int vari_, std::string varstr_) est utilisé pour  
    // initialiser la valeur
```

La variable peut aussi être explicitement initialisée à l'état invalide :

```
boost::optional<int> opt_int(boost::none);
```

Les différentes initialisations correspondent à des comportements différents :

```
boost::optional<CMaClasse> opt_ma_classe_1;  
    // Aucun constructeur de CMaClasse n'est appelé  
    // opt_ma_classe_1 est dans l'état invalide  
  
boost::optional<CMaClasse> opt_ma_class_2(boost::none);
```

```
// Aucun constructeur de CMaClasse n'est appelé
// opt_ma_classe_2 est dans l'état invalide

boost::optional<CMaClasse> opt_ma_class_3(CMaClasse(1,"chaîne"));
// Le constructeur par copie CMaClasse::CMaClasse(CMaClasse const&) est appelé
// opt_ma_classe_3 contient une valeur valide

boost::optional<CMaClasse> opt_ma_class_4(boost::in_place(1,"chaîne"));
// Le constructeur adéquat CMaClasse::CMaClasse(int,std::string) est appelé
// opt_ma_classe_4 contient une valeur valide
```

V-C - Affecter une valeur

L'opérateur d'affectation permet de positionner la valeur d'une variable boost::optional :

```
boost::optional<int> opt_int;
opt_int = 3; // Initialisation avec une valeur

boost::optional<CMaClasse> opt_ma_classe;
opt_ma_classe = boost::in_place(1,"chaîne"); // affectation avec un constructeur sur place

boost::optional<CMaClasse> opt_ma_classe_2;
opt_ma_classe_2 = CMaClasse(1,"chaîne"); // affectation avec un constructeur par copie

boost::optional<CMaClasse> opt_ma_classe_3;
opt_ma_classe_3 = boost::none; // on force l'état invalide
```

Le comportement de l'opérateur d'affectation diffère selon l'état de la variable :

```
boost::optional<CMaClasse> opt_ma_classe;
opt_ma_classe = CMaClasse(); // Appel du constructeur par copie
// pour une variable dans l'état non initialisé
// ...
opt_ma_classe = CMaClasse(); // Appel de l'opérateur '=' de CMaClasse
// pour une variable dans l'état initialisé

boost::optional<CMaClasse> opt_ma_classe2(CMaClasse());
opt_ma_classe2 = boost::none; // Appel du destructeur de CMaClasse
// pour une variable initialisé.
// ...
opt_ma_classe2 = boost::none; // Aucun appel pour une variable
// dans l'état non initialisé

boost::optional<CMaClasse> opt_ma_classe3;
opt_ma_classe3 = boost::in_place(1,"chaîne"); // Appel du constructeur adéquat
// CMaClasse::CMaClasse(int, std::string) pour une variable
// dans l'état non initialisé
// ...
opt_ma_classe3 = boost::in_place(42,"Autre chaîne"); // Appel du destructeur
// puis de CMaClasse::CMaClasse(int, std::string) pour une variable
// dans l'état initialisé
```

V-D - Vérifier l'état d'une variable

Une variable boost::optional peut être utilisée comme un booléen indiquant son état :

```
boost::optional<int> ma_variable(4);
if(ma_variable){
    std::cout<<"La variable est initialisée";
}
ma_variable = boost::none;
bool est_initialise = ma_variable;
if(!est_initialise){
    std::cout<<"La variable ne contient plus de valeur valide";
}
```

L'opérateur '!' donne la valeur inverse :

```
boost::optional<std::string> opt_string;
if(!opt_string){
    opt_string = "initialise";
}
if(!opt_string){
    std::cout<<"La variable n'est pas initialisée ?";
}
```

V-E - Récupérer la valeur

Une fois la variable boost::optional initialisée, la valeur est simplement récupérée avec l'opérateur '*' :

```
boost::optional<int> opt_int(3);
if(opt_int){
    std::cout<<"La valeur est : "<< *opt_int;
}
```

Attention, l'accès à la valeur d'une variable dans l'état invalide aboutit à un comportement indéterminé :

```
boost::optional<int> opt_int;
std::cout<<"La valeur est : "<< *opt_int; // plantage assuré !
```

Il faut bien faire attention à ne pas oublier cette étoile magique sans quoi le compilateur cherche à évaluer la variable comme booléen :

```
boost::optional<int> opt_int;
std::cout<<opt_int<<std::endl; // Affiche 'faux' ou '0'
opt_int = 42;
std::cout<<opt_int<<std::endl; // Affiche 'vrai' ou '1'
std::cout<<*opt_int<<std::endl; // Affiche 42
```

Il faut bien distinguer les deux écritures différentes :

```
boost::optional<int> opt_int;
opt_int = 42; // on initialise la variable opt_int
*opt_int = 54; // on récupère l'entier associé et on en change sa valeur
```

La différence est plus flagrante avec une classe :

```
boost::optional<CMaClasse> opt_ma_classe;
opt_ma_classe = CMaClasse(); // Initialisation de opt_ma_classe avec un constructeur par copie
*opt_ma_classe = CMaClasse(); // La valeur est récupérée et l'opérateur d'affectation '='
// de CMaClasse est appelé
```

Une fois déréférencé, l'objet contenu s'utilise comme tous les autres objets du même type :

```
boost::optional<CMaClasse> opt_ma_classe(CMaClasse());
(*opt_ma_classe).MaMethode(); // Appel à CMaClasse::MaMethode
```

La surcharge de l'opérateur '->' permet l'utilisation d'obtenir un pointeur sur l'objet contenu :

```
boost::optional<CMaClasse> opt_ma_classe(CMaClasse());
opt_ma_classe->MaMethode(); // Appel à CMaClasse::MaMethode
```

Comme l'opérateur '*', l'appel à '->' sur un objet non initialisé provoque un comportement indéfini :

```
boost::optional<CMaClasse> opt_ma_classe;
opt_ma_classe->MaMethode();// plantage assuré !
```

Pour terminer le tour de la question, signalons l'existence des fonctions équivalentes aux opérateurs décrits ci-dessus :

```
boost::optional<CMaClasse> opt_ma_classe(CMaClasse);
opt_ma_classe.get().MaMethode();
opt_ma_classe.get_ptr()->MaMethode();
```

get_ptr présente la particularité de ne pas planter si la variable n'a pas été initialisée mais retourne un pointeur NULL :

```
boost::optional<CMaClasse> opt_ma_classe;
if(opt_ma_classe.get_ptr()==NULL){
    opt_ma_classe = CMaClasse();
}
```

V-F - Comparer deux variables boost::optional

L'opérateur '==' permet de comparer deux variables boost::optional de même type :

```
boost::optional<int> opt_int1(1);
boost::optional<int> opt_int2(1);
if(opt_int1==opt_int2){
    ++*opt_int2;
}
```

La comparaison de deux variables boost::optional se fait sur leur état, et si besoin, sur leur valeur. Ainsi l'opérateur '==' du type de base est appelé si les deux opérandes sont dans l'état initialisé :

operator==	Etat indéterminé	Etat déterminé
Etat indéterminé	vrai	faux
Etat déterminé	faux	(*opt1)==(*opt2)

Notons que les opérateurs '!=', '<', '>', '<=' et '>=' sont aussi redéfinis. L'état indéterminé est considéré comme plus petit que l'état déterminé. Ensuite, si les deux opérandes sont dans l'état déterminé, c'est l'opérateur de la classe de base qui est invoqué :

operator <	Etat indéterminé	Etat déterminé
Etat indéterminé	faux (inférieur strict)	vrai
Etat déterminé	faux	(*opt1)<(*opt2)

V-G - Utilisation dans un conteneur

On peut utiliser boost::optional comme n'importe quel type :

```
std::vector<boost::optional<CMaClasse> > vect_opt_ma_classe;
vect_opt_ma_classe.push_back(CMaClasse(1,"un"));
vect_opt_ma_classe.push_back(CMaClasse(2,"deux"));
vect_opt_ma_classe.push_back(CMaClasse(3,"trois"));
```

Attention, un élément du vecteur est de type boost::optional et non de type CMaClasse. Il faut déréférencer l'objet boost::optional pour retrouver l'élément contenu en veillant bien sûr à ce que l'objet soit dans l'état initialisé :

```
void Fonction(boost::optional<CMaClasse> const & opt_)
{
    if(opt_) {
```



```

    opt_ -> MaMethode();
}
}
int main()
{
    std::vector<boost::optional<CMaClasse> > vect_opt_ma_classe;
    vect_opt_ma_classe.push_back(CMaClasse(1, "un"));
    vect_opt_ma_classe.push_back(CMaClasse(2, "deux"));
    vect_opt_ma_classe.push_back(CMaClasse(3, "trois"));

    std::for_each(vect_opt_ma_classe.begin(), vect_opt_ma_classe.end(), Fonction);

    return 0;
}

```

VI - Quelques exemples

VI-A - Valeur en retour indéterminée

Commençons par un exemple de fonction s'appuyant sur `boost::optional` pour gérer une valeur en retour signifiant que le résultat n'a pu être déterminé :

```

boost::optional<Client> CMaBase::TrouverExact(std::string const &nom_) const
{
    if (ExisteClient(nom_)) {
        return boost::optional<Client>(LireClient(nom_)); // on retourne le client
    }

    return boost::optional<Client>(); // on retourne une instance non initialisée !
}

bool CMaBase::ExisteClient(std::string const &nom_) const
{
    // ...
}

Client CMaBase::LireClient(std::string const &nom_) const
{
    // ...
}

```

Lorsqu'il existe une entrée dont le nom coïncide avec celui donné en paramètre, alors la valeur en retour est initialisée avec l'enregistrement lu. Sinon, la méthode retourne une instance de `boost::optional` dans l'état non initialisé. L'utilisation est alors :

```

bool CVueClient::DemanderAffichageFiche()
{
    std::string nom = DemanderNom();
    boost::optional<Client> le_client(ma_base.TrouverExact(nom));
    bool b_trouve = le_client; // prend la valeur vrai si la variable contient une valeur
    if (b_trouve) {
        // Affichage de client
        AffichageFiche(*le_client); // on déréférence pour obtenir la valeur
    }
    else {
        AvertirOperateur(nom, ERR_Client_Inconu);
    }
    return b_trouve;
}

void CVueClient::AffichageFiche(Client const &le_client_)
{
    // procédure d'affichage...
}

```

VI-B - Variable locale non initialisée

Une variable `boost::optional` peut simplifier l'écriture d'une méthode en tirant bénéfice de l'état non initialisé.

```
void DlgAvancement::OnEvtAvancement(Evenement evt_)
{
    boost::optional<bool> optAffichage;
    switch(evt_.type) {
    case Evenement::E_Demarrage:
        SetRange(evt_.min, evt_.max);
        optAffichage = true;
        break;
    case Evenement::E_Avancement:
        StepIt();
        optAffichage = boost::none;
        break;
    case Evenement::E_Termine:
        SetPosition(evt_.max);
        optAffichage = false;
        break;
    default:
        ASSERT(false);
        break;
    }
    if(optAffichage){ // on teste si la valeur est initialisée.
        Show(*optAffichage); // on déréférence pour obtenir la valeur booléenne.
        UpdateControles();
        Layout();
    }
}
```

Dans cet exemple, on veut effectuer un ensemble d'action pour deux cas précis. Pour les autres cas, la variable `boost::optional` utilise l'état non initialisé.

VI-C - Membre d'une classe optionnel

Un membre peut ne pas avoir de valeur. `boost::optional` évite l'utilisation d'un pointeur avec tous les risques que cela peut comporter :

```
struct Adresse
{
    int numero;
    std::string nom_rue;
    std::string code_postal;
    std::string ville;
    boost::optional<std::string> cedex;
};
```

L'état invalide signifie que l'adresse ne possède pas de cedex.

Une autre utilisation d'un membre `boost::optional` se présente dès qu'un membre a une durée de vie inférieure à celle de la classe le contenant. `boost::optional` gère la composition sans pointeur :

```
class wxMaVue
{
    ...
protected:
    boost::optional<wxTimer> m_timer;
};

wxMaVue::DemarrerTimer()
{
    m_timer = boost::in_place<wxTimer>(); // si le membre était initialisé,
```

```
// le destructeur est appelé arrêtant ainsi l'ancien timer
// un nouveau timer est créé
m_timer->Start(m_intervalle);
...
}

wxMaVue::ArreterTimer()
{
    m_timer = boost::none; // si m_timer est initialisé
    // le destructeur est appelé arrêtant le timer en cours
    ...
}
```

VI-D - Réassigner une référence

L'utilisation d'une référence avec boost::optional est possible. Dès lors, la réinitialisation de la variable correspond à une réassignation de référence :

```
int main()
{
    int val_1 (5);
    int val_2 (42);

    boost::optional<int&> opt_ri;
    opt_ri = val_1; // opt_ri est liée à val_1
    std::cout<<val_1<<" "<<*opt_ri<<" "<<val_2<<std::endl;
    *opt_ri = 12; // on change la valeur de val_1
    std::cout<<val_1<<" "<<*opt_ri<<" "<<val_2<<std::endl;
    *opt_ri = val_2; // on change la valeur de val_1 qui vaut maintenant celle de val_2
    std::cout<<val_1<<" "<<*opt_ri<<" "<<val_2<<std::endl;
    opt_ri = val_2; // on réassigne opt_ri : opt_ri est liée à val_2
    *opt_ri = 5; // on change la valeur de val_2
    std::cout<<val_1<<" "<<*opt_ri<<" "<<val_2<<std::endl;

    return 0;
}
```

VII - Description de la bibliothèque

VII-A - En-tête

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

- boost/optional.hpp
- boost/optional/optional.hpp
- boost/none.hpp

Concernant les deux premiers en-têtes, le principe de Boost veut qu'un en-tête global existe sous la racine de 'boost/' pour une bibliothèque. Les en-têtes présents dans un répertoire particulier permettent de n'utiliser qu'une partie spécifique de la bibliothèque. Boost.Optional ne fournissant qu'un seul service, l'en-tête à la racine se contente d'inclure le second en-tête.

L'utilisation du troisième en-tête est expliquée ci-dessous.

VII-B - Espace de nommage

L'espace de nommage est :

```
namespace boost
```

VII-C - Interface

VII-C-1 - La classe

Classe optional :

```
template<class T>
class optional
{
    public :
        // Constructeurs :
        optional () ;
        optional ( boost::none_t ) ;
        optional ( T const& v ) ;
        optional ( bool condition, T const& v ) ;
        optional ( optional const& rhs ) ;
        template<class U> explicit optional ( optional<U> const& rhs ) ;
        template<class InPlaceFactory> explicit optional ( InPlaceFactory const& f ) ;
        template<class TypedInPlaceFactory> explicit optional ( TypedInPlaceFactory const& f ) ;

        // Opérateur d'affectation :
        optional& operator = ( boost::none_t ) ;
        optional& operator = ( T const& v ) ;
        optional& operator = ( optional const& rhs ) ;
        template<class U> optional& operator = ( optional<U> const& rhs ) ;
        template<class InPlaceFactory> optional& operator = ( InPlaceFactory const& f ) ;
        template<class TypedInPlaceFactory> optional& operator = ( TypedInPlaceFactory const& f ) ;

        // Accesseur
        T const& get() const ;
        T& get() ;
        T const& get_value_or( T const& default ) const ;
        T const& operator *() const ;
        T& operator *() ;
        T const* get_ptr() const ;
        T* get_ptr() ;
        T const* operator ->() const ;
        T* operator ->() ;

        // Diagnostic de l'état
        operator unspecified-bool-type() const ;
        bool operator !() const ;
};
```

T peut être un type ou une référence :

Exemple de variable optional :

```
boost::optional<int> mon_entier;
boost::optional<CMaRessource&> ma_ressource;
```

VII-C-2 - Convention

Certaines définitions sont légèrement différentes selon que la classe est instanciée avec une référence ou pas. Les définitions ci-dessous utilisent par convention la notation suivante :

- optional<T> : la définition s'applique sur le type (optional<T>) comme sur une référence (optional<T&>) ;
- optional<non_ref(T)> : la définition s'applique pour le type (optional<T>) mais pas sur une référence (optional<T&>) ;
- optional<T&> : la définition s'applique pour une référence (optional<T&>) mais pas pour le type (optional<T>).

VII-C-3 - Les constructeurs

- Constructeur par défaut :

```
optional<T>::optional();
```

 - **Objectif** : Constructeur par défaut.
 - **Post-condition** : L'état est non-initialisé.
 - **Exception(s)** : Aucune.
 - **Exemple** :

```
optional<T> ma_variable;
```
 - **Notes** : Le constructeur de T *n'est pas* appelé.
- Constructeur non-initialisé :

```
optional<T>::optional(boost::none_t);
```

 - **Objectif** : Constructeur d'un objet non initialisé.
 - **Post-condition** : L'état est non-initialisé.
 - **Exception(s)** : Aucune.
 - **Exemple** :

```
optional<T> ma_variable(boost::none);
```
 - **Notes 1** : Le constructeur de T *n'est pas* appelé.
 - **Notes 2** : `boost::none` de type `boost::none_t` est défini dans `boost/none.hpp`.
- Constructeur initialisé :

```
optional<non_ref(T)>::optional(T const& _copie);
```

 - **Objectif** : Constructeur d'un objet initialisé.
 - **Post-condition** : L'état est initialisé et la valeur est une copie du paramètre.
 - **Exception(s)** : Celle(s) que lève le constructeur par copie de `T::T(T const&)`;
 - **Exemple** :

```
T mon_element;
optional<T> ma_variable(mon_element);
```
 - **Notes** : Le constructeur `T::T(T const&)` est appelé.
- Constructeur initialisé :

```
optional<T&>::optional(T & _reference);
```

 - **Objectif** : Constructeur d'un objet initialisé.
 - **Post-condition** : L'état est initialisé et la valeur est la référence fournie en paramètre. Il s'agit bien de la valeur de la référence et non une copie de l'objet.
 - **Exception(s)** : Aucune.
 - **Exemple** :

```
T mon_element;
T& ma_reference = mon_element;
optional<T&> ma_variable(ma_reference);
```
 - **Notes** : Le constructeur de T *n'est pas* appelé.
- Constructeurs conditionnels :

```
optional<non_ref(T)>::optional(bool _b_initialise, T const& _copie);
optional<T&>::optional(bool _b_initialise, T & _reference);
```

 - **Objectif** : Constructeurs conditionnels.
 - **Post-condition** : Si `_b_initialise` s'évalue à `false`, l'état est non-initialisé. Si `_b_initialise` s'évalue à `true`, l'état est initialisé ; la valeur est une copie du paramètre pour le premier cas, une copie de la référence dans le second.
 - **Exception(s)** : cf. Note.
 - **Exemple** :

```
T mon_element;
T& ma_reference = mon_element;
optional<T&> ma_variable(true, ma_reference);
optional<T> mon_autre_variable(false, mon_element);
```

- **Notes** : Si la condition s'évalue à *true*, alors ces constructeurs sont équivalents à `optional<non_ref(T)>::optional(T const& _copie)` respectivement `optional<T>::optional(T & _reference)`. Si la condition vaut *false*, alors les constructeurs sont équivalents à `optional<T>::optional(boost::none_t)`.

Constructeur par copie :

```
optional<non_ref(T)>::optional(optional const& _copie);
```

- **Objectif** : Constructeur par copie.
- **Post-condition** : Si l'état de `_copie` est initialisé, alors l'état de `*this` est initialisé et la valeur est une copie de la valeur de `_copie`. Si l'état de `_copie` est non-initialisé, alors l'état de `*this` est non-initialisé.
- **Exception(s)** : Si l'état de `_copie` est initialisé, alors les exceptions levées sont celles de `T::T(T const &)`.
- **Exemple** :

```
T mon_element;
optional<T> ma_variable(mon_element);
optional<T> mon_autre_variable(ma_variable);
```

- **Notes** : Si l'état de `_copie` est initialisé, alors le constructeur de copie de `T` est appelé.

Constructeur par copie :

```
optional<T&>::optional(optional const& _copie);
```

- **Objectif** : Constructeur par copie.
- **Post-condition** : Si l'état de `_copie` est initialisée, alors l'état de `*this` est initialisé et la valeur est la même référence que `_copie`. Si l'état de `_copie` est non-initialisé, alors l'état de `*this` est non-initialisé.
- **Exception(s)** : Aucune.
- **Exemple** :

```
T mon_element;
T& ma_reference = mon_element;
optional<T&> ma_variable(ma_reference);
optional<T&> mon_autre_variable(ma_variable);
```

- **Notes** : Le constructeur de `T` n'est pas appelé.

Constructeur de conversion :

```
template<class U>
optional<non_ref(T)>::optional(optional<U> const& _copie);
```

- **Objectif** : Constructeur de conversion.
- **Post-condition** : Si l'état de `_copie` est initialisé, alors l'état de `*this` est initialisé et sa valeur est une copie de la valeur de `_copie` convertie en `T` : `T::T(U const &)`. Si l'état de `_copie` n'est pas initialisé, alors l'état de `*this` n'est pas initialisé.
- **Exception(s)** : Si l'état de `_copie` est initialisé, alors les exceptions levées sont celles de `T::T(U const &)`.
- **Exemple** :

```
U mon_element;
optional<U> ma_variable(mon_element);
optional<T> mon_autre_variable(ma_variable);
```


- **Notes** : Si l'état de `_copie` est initialisé, alors le constructeur `T::T(U const &)` est appelé.

Constructeur In Place Factory :

```
template<class InPlaceFactoryT>
optional<non_ref(T)>::optional(InPlaceFactoryT _usine);
template<class TypedInPlaceFactoryT>
optional<non_ref(T)>::optional(TypedInPlaceFactoryT _usine);
```

- **Objectif** : Constructeur avec un objet construit en place.
- **Post-condition** : l'état de `*this` est initialisé et sa valeur est celle résultant de sa construction avec l'usine.
- **Exception(s)** : Celle(s) levée(s) par le constructeur appelé par l'usine.
- **Exemple** :

```
optional<T> ma_variable(boost::in_place(p0, p1, p2));
optional<T> mon_autre_variable(boost::in_place<T>(p0,p1,p2));
```

- **Notes :** Pour plus d'informations sur la bibliothèque Boost.In Place factory, vous pouvez consulter : 
Tutoriel Boost.In Place Factory

VII-C-4 - Surcharges de l'opérateur '='

Les différentes surcharges de l'opérateur d'affectation sont en cohérence avec celles des différents constructeurs.

- **Réinitialisation :**

```
optional<T>& optional<T>::operator= ( boost::none_t );
```

 - **Objectif :** Réinitialisation de l'objet.
 - **Post-condition :** L'état est non initialisé.
 - **Exception(s) :** Aucune.
 - **Exemple :**

```
T mon_element;
optional<T> ma_variable(mon_element);
ma_variable = boost::none;
```
 - **Notes 1 :** Lorsque T n'est pas une référence et si l'état était initialisé, alors le destructeur de T est appelé sur la valeur précédente.
 - **Notes 2 :** La bibliothèque présuppose que le destructeur de T ne lève pas d'exception.
- **Affectation d'un objet :**


```
optional<non_ref(T)>& optional<non_ref(T)>::operator= ( T const& _valeur );
```

 - **Objectif :** Affectation d'un élément.
 - **Post-condition :** L'état est initialisé et la valeur est une copie du paramètre.
 - **Exception(s) :** Si l'état était déjà initialisé, celle(s) que lève l'opérateur T::operator=(T const &), sinon celle(s) que lève le constructeur par copie T::T(T const &).
 - **Exemple :**

```
T mon_element;
optional<T> ma_variable;
ma_variable = mon_element;
T autre_element;
ma_variable = autre_element;
```
 - **Notes :** Si l'état est initialisé, et que l'opérateur T::operator= lève une exception, l'état reste initialisé. Il est du ressort de l'opérateur d'affectation de T de gérer correctement les exceptions pour que la valeur soit dans un état valide.
Si l'état n'est pas initialisé, et que le constructeur par copie de T lève une exception alors l'état reste non-initialisé et la valeur est indéterminée.
- **Affectation d'une référence :**

```
optional<T&>& optional<T&>::operator= ( T& const& _reference );
```

 - **Objectif :** Affectation d'une référence.
 - **Post-condition :** L'état est initialisé et la valeur est la référence.
 - **Exception(s) :** Aucune.
 - **Exemple :**

```
T mon_element;
optional<T&> ma_variable;
ma_variable = mon_element;
T autre_element;
ma_variable = autre_element;
```
 - **Notes :** C'est bien un changement de référence : si l'état était déjà initialisé, l'objet se trouve lié à la nouvelle référence ! Ni l'opérateur d'affectation de T, ni son constructeur par copie ne sont appelés.
 *Grâce à cette surcharge, on peut réassigner une référence chose impossible sinon :*

```
int i_1(1);
int i_2(2);
```

```
int &ri_1 = i_1;
int &ri_2 = i_2;
ri_1 = ri_2; //Change la valeur de i_1
              // i_1 vaut 2 et i_2 vaut 2 !

i_1 = 1;
i_2 = 2;
boost::optional<int> opt_1(i_1);
opt_1 = ri_2; // maintenant opt_1 fait référence à i_2 ! i_1 n'est pas changé.
              // i_1 vaut 1 et i_2 vaut 2
```

Affectation d'un boost::optional :

```
optional& optional<non_ref(T)>::operator= ( optional const& _copie );
```

- **Objectif :** Affectation d'une autre instance de *boost::optional*.
- **Post-condition :** Prend l'état de *_copie* et si nécessaire, sa valeur.
- **Exception(s) :** Si l'état de *_copie* est non-initialisé, alors aucune exception n'est levée. Si l'état de *_copie* est initialisé et l'état de *this* n'est pas initialisé, alors les exceptions levées sont celles du constructeur par copie de T. Si l'état de *_copie* est initialisé et l'état de *this* est initialisé, alors les exceptions levées sont celles de l'opérateur d'affectation de T.

Exemple :

```
T mon_element;
boost::optional<T> ma_variable(mon_element);
boost::optional<T> mon_autre_variable;
mon_autre_variable = ma_variable;
```

- **Notes :** Si *this* passe de l'état initialisé à non initialisé, alors le destructeur de T est appelé sur la valeur précédente.

Affectation d'un boost::optional, référence :

```
optional<T&>& optional<T&>::operator= ( optional<T&>& _reference );
```

- **Objectif :** Affectation d'un boost::optional (référence).
- **Post-condition :** Prend l'état de *_copie* et si nécessaire, sa référence.
- **Exception(s) :** Aucune.

Exemple :

```
T mon_element;
T mon_second_element;
boost::optional<T&> ma_variable(mon_element);
boost::optional<T&> mon_autre_variable(mon_second_element);
mon_autre_variable = ma_variable;
```

- **Notes :** Voir l'opérateur d'affectation d'une référence : *optional<T&>::operator=(T& const& _reference)* ci dessus.

Affectation d'un optional avec conversion :

```
template<class U>
optional<T&>& optional<non_ref(T)&>::operator= ( optional<U&>& _copie );
```

- **Objectif :** Affectation d'un optional avec conversion de la valeur.
- **Post-condition :** Prend l'état de *_copie* et, si nécessaire, sa valeur est initialisée avec une conversion de la valeur *T::T(U const &)*.
- **Exception(s) :** Si l'état de *_copie* est non-initialisé, alors aucune exception n'est levée. Si l'état de *_copie* est initialisé et l'état de *this* est non-initialisé, alors les exceptions levées sont celles du constructeur de conversion de T : *T::T(U const &)*. Si l'état de *_copie* est initialisé et l'état de *this* est initialisé, alors les exceptions levées sont celles de l'opérateur d'affectation de T : *T::operator=(U const &)*.

Exemple :

```
T mon_element;
U mon_second_element;
boost::optional<T;> ma_variable(mon_element);
boost::optional<U;> mon_autre_variable(mon_second_element);
ma_variable = mon_autre_variable;
```

- **Notes :** Si *this* passe de l'état initialisé à non initialisé, alors le destructeur de T est appelé sur la valeur précédente. La bibliothèque présuppose que le destructeur de T ne lève pas d'exception.

Affectation avec construction en place :

```
template<class InPlaceFactory>
optional<T>& optional<non_ref(T)>::operator= (InPlaceFactory const&_usine);
template<class TypedInPlaceFactory>
optional<T>& optional<non_ref(T)>::operator= (TypedInPlaceFactory const&_usine );
```

- **Objectif** : Affectation d'un optional avec construction en place.
- **Post-condition** : L'état de **this* est initialisé et sa valeur est celle résultant de sa construction avec l'usine.
- **Exception(s)** : Celle(s) levée(s) avec le constructeur appelé par l'usine.
- **Exemple** :

```
T mon_element;
mon_element = boost::in_place(p0, p1, p2);
mon_element = boost::in_place<T>(p0, p1, p2);
```

- **Notes 1** : Si l'état était initialisé, le destructeur est appelé sur la valeur précédente, puis l'usine est appelée pour construire l'objet avec le constructeur pris en charge par cette dernière. Il n'y a pas d'appel à l'opérateur d'affectation ou au constructeur par copie avec une valeur temporaire construite par l'usine. La bibliothèque présuppose que le destructeur de T ne lève pas d'exception.
- **Notes 2** : Pour plus d'informations sur la bibliothèque Boost.In Place factory, vous pouvez consulter :

 **Tutoriel Boost.In Place Factory**

VII-C-5 - Les accesseurs

VII-C-5-a - Par valeur

Accesseurs sur la valeur :

```
T const& optional<non_ref(T)>::get() const;
T& optional<non_ref(T)>::get();
T const& optional<non_ref(T)>::operator *() const;
T& optional<non_ref(T)>::operator *();
```

- **Objectif** : Récupérer la valeur initialisée.
- **Post-condition** : Retourne la valeur de l'objet. Chacune des fonctions est déclinée en version constante ou non.
- **Exception(s)** : Aucune.
- **Exemple** :

```
int i_1(1);
boost::optional<int> opt_1(i_1);
int &ri_1 = *opt_1;
ri_1 = 33;
std::cout<<"i_1 = "<<i_1<<std::endl; // 1
std::cout<<"opt_1 = "<<*opt_1<<std::endl; // 33
std::cout<<"ri_1 = "<<ri_1<<std::endl; // 33
```

- **Notes** : En mode debug, BOOST_ASSERT est utilisé pour s'assurer que l'objet est initialisé. Attention, *boost::optional* ne lève pas d'exception si l'état est non initialisé. L'assertion est là pour aider en mode debug.
Ce qu'il faut comprendre à travers ce comportement, c'est la volonté des auteurs de rester cohérents avec la norme qui dit que l'utilisation d'une variable non initialisée est indéterminée. En ce sens, l'utilisation d'une instance de *boost::optional* dans l'état non initialisé ne provoque pas d'erreur (pas d'exception) mais maintient un comportement indéterminé puisque la valeur effectivement retournée n'est pas initialisée.

Accesseurs sur la valeur (référence) :

```
T const& optional<T>::get() const;
T& optional<T>::get();
T const& optional<T>::operator *() const;
T& optional<T>::operator *();
```

- **Objectif** : Récupérer la référence initialisée.

- **Post-condition** : Retourne la référence de l'objet. Chacune des fonctions est déclinée en version constante ou non.
- **Exception(s)** : Aucune.
- **Exemple** :

```
int i_1(1);
boost::optional<int> opt_1(i_1);
int &ri_1 = *opt_1;
ri_1 = 33;
std::cout<<"i_1 = "<<i_1<<std::endl; // 33
std::cout<<"opt_1 = "<<*opt_1<<std::endl; // 33
std::cout<<"ri_1 = "<<ri_1<<std::endl; // 33
```

- **Notes** : Le comportement est le même que précédemment pour une valeur non initialisée. Notons toutefois qu'à l'instar d'un pointeur, l'utilisation d'une référence non initialisée, donc non valide, aboutit rapidement à une erreur d'adressage !

Accesseur avec valeur par défaut :

```
T const& get_value_or( T const& _default) const;
T& get_value_or( T & _default);
```

- **Objectif** : Rendre l'accesseur déterminé quelque soit l'état.
- **Post-condition** : Si la variable est initialisée, alors le comportement est identique à `optional::get`. Sinon, la valeur retournée est celle fournie en paramètre.
- **Exception(s)** : Aucune.
- **Exemple** :

```
int i_1(1);
boost::optional<int> opt_1(i_1);
std::cout<<opt_1.get_value_or(20)<<std::endl; // 1

int i_2(12);
boost::optional<int> opt_2;
std::cout<<opt_2.get_value_or(i_2)<<std::endl; // 12
```

- **Notes** : La méthode est déclinée en version `const` et non `const`. Cette méthode permet d'avoir toujours une valeur déterminée en retour.

VII-C-5-b - Par adresse

Accesseurs de l'adresse :

```
T const* optional<non_ref(T)>::get_ptr() const;
T* optional<non_ref(T)>::get_ptr();
```

- **Objectif** : Récupérer l'adresse de la valeur.
- **Post-condition** : Retourne l'adresse de la valeur si l'élément est initialisé, NULL sinon.
- **Exception(s)** : Aucune.
- **Exemple** :

```
int i_1(1);
boost::optional<int> opt_1(i_1);
int &ri_1 = *(opt_1.get_ptr());
ri_1 = 33;
std::cout<<"i_1 = "<<i_1<<std::endl; // 1
std::cout<<"opt_1 = "<<*(opt_1.get_ptr())<<std::endl; // 33
std::cout<<"ri_1 = "<<ri_1<<std::endl; // 33
```

- **Notes** : On note que pour une variable non initialisée, le comportement reste déterminé puisque la valeur retournée est NULL. Comme la valeur en retour est toujours déterminée, aucune assertion n'est utilisée pour vérifier l'état.

Accesseurs de l'adresse :

```
T const* optional<T>::get_ptr() const;
T* optional<T>::get_ptr();
```

- **Objectif** : Récupérer l'adresse de la référence.
- **Post-condition** : Retourne l'adresse de la référence si l'élément est initialisé, NULL sinon.
- **Exception(s)** : Aucune.

- **Exemple :**

```
int i_1(1);
boost::optional<int> opt_1(i_1);
int &ri_1 = *(opt_1.get_ptr());
ri_1 = 33;
std::cout<<"i_1 = "<<i_1<<std::endl; // 33
std::cout<<"opt_1 = "<<*(opt_1.get_ptr())<<std::endl; // 33
std::cout<<"ri_1 = "<<ri_1<<std::endl; // 33
```

- **Notes :** On note que pour une variable non initialisée, le comportement reste déterminé puisque la valeur retournée est NULL. Comme la valeur en retour est toujours déterminée, aucune assertion n'est utilisée pour vérifier l'état.

Voici un exemple de code pour bien comprendre la différence de comportement entre l'utilisation avec un type et l'utilisation avec une référence :

Utilisation non_ref :

```
int i_1(1);
boost::optional<int> opt_1(i_1);
int &ri_1 = *(opt_1.get_ptr());
int &ri_2 = i_1;
std::cout<<std::boolalpha;
std::cout<< (&ri_1==opt_1.get_ptr())<<std::endl; // true
std::cout<< (&ri_2==&i_1)<<std::endl; // true
std::cout<< (&ri_1==&i_1)<<std::endl; // false
std::cout<< (&ri_2==opt_1.get_ptr())<<std::endl; // false
std::cout<< (&ri_2==&ri_1)<<std::endl; // false
std::cout<< (&i_1==opt_1.get_ptr())<<std::endl; // false
```

La valeur de la variable interne de *opt_1* est 1.
ri_1 'pointe' sur la variable interne de *opt_1* contenant la valeur.
ri_2 'pointe' sur *i_1*.

Utilisation ref :

```
int i_1(1);
boost::optional<int&> opt_1(i_1);
int &ri_1 = *(opt_1.get_ptr());
int &ri_2 = i_1;
std::cout<<std::boolalpha;
std::cout<< (&ri_1==opt_1.get_ptr())<<std::endl; // true
std::cout<< (&ri_2==&i_1)<<std::endl; // true
std::cout<< (&ri_1==&i_1)<<std::endl; // true
std::cout<< (&ri_2==opt_1.get_ptr())<<std::endl; // true
std::cout<< (&ri_2==&ri_1)<<std::endl; // true
std::cout<< (&i_1==opt_1.get_ptr())<<std::endl; // true
```

La variable interne de *opt_1* contient l'adresse de *i_1*.
ri_1 'pointe' sur *i_1*.
ri_2 'pointe' sur *i_1*.

- **Surcharge de l'opérateur -> :**

```
T const* optional<T>::operator ->() const ;
T* optional<T>::operator ->() ;
```

- **Objectif :** Ces surcharges sont strictement équivalentes à *optional::get_ptr*.
- **Post-condition :** Voir ci-dessus la description de *optional::get_ptr*.
- **Exception(s) :** Voir ci-dessus la description de *optional::get_ptr*.

VII-C-6 - Vérification de l'état

- **Conversion :**

```
operator unspecified-bool-type() const ;
```

- **Objectif** : Vérifier l'état initialisé.
- **Post-condition** : La valeur retournée s'évalue à *true* si l'objet est dans l'état initialisé et à *false* sinon.
- **Exception(s)** : Aucune.
- **Exemple** :

```
boost::optional<T> opt;
std::cout<<std::boolalpha;
bool b_initialise = opt;
std::cout<<b_initialise<<std::endl; // false
opt = T();
if(opt){
    std::cout<<"OK"<<std::endl;
}
```

Vérification état non initialisé :

```
bool optional<T>::operator!() const;
```

- **Objectif** : Vérifier l'état non initialisé.
- **Post-condition** : La valeur retournée est *true* si l'objet n'est pas initialisé, *false* sinon.
- **Exception(s)** : Aucune.
- **Exemple** :

```
boost::optional<T> opt;

std::cout<<std::boolalpha;
bool b_initialise = !opt;
std::cout<<b_initialise<<std::endl; // true
opt = T();
if(!opt){
    std::cout<<"Ah bon ?"<<std::endl;
}
else{
    std::cout<<"OK"<<std::endl;
}
```

VII-C-7 - Méthodes obsolètes

Les méthodes suivantes sont définies dans la classe Boost.Optional. Cependant, la documentation officielle les déclare comme obsolètes. Une méthode alternative est proposée pour retrouver le service précédemment fourni :

Méthode obsolète	Nouvelle méthode
<code>bool optional<T>::is_initialized() const;</code>	<code>operator unspecified-bool-type() const ;</code> <code>bool optional<T>::operator!() const;</code>
<code>void optional<T>::reset();</code>	<code>optional<T>& optional<T>::operator=(boost::none_t);</code>
<code>void optional<T>::reset (T const&);</code>	<code>optional<non_ref(T)>& optional<non_ref(T)>::operator= (T const&_valeur);</code> <code>optional<T&>& optional<T&>::operator= (T& const&_reference);</code>

VII-C-8 - Accesseurs externes

Des accesseurs sont définis en dehors de la classe. Ils se contentent d'appeler la méthode correspondante sur l'objet fourni en paramètre. Le tableau suivant récapitule ces accesseurs avec leur équivalent :

Accesseur externe	Méthode invoquée
<code>template<class T></code>	<code>T& optional<T>::get() const;</code>

<code>inline T const& get(optional<T> const& _opt);</code>	
<code>template<class T> inline T & get(optional<T> & _opt);</code>	<code>T& optional<T>::get();</code>
<code>template<class T> inline T const* get (optional<T> const* _p_opt); template<class T> inline T const* get_pointer(optional<T> const& _p_opt);</code>	<code>T const* optional<T>::get_ptr() const;</code>
<code>template<class T> inline T * get (optional<T> * _p_opt); template<class T> inline T * get_pointer(optional<T> * _p_opt);</code>	<code>T * optional<T>::get_ptr();</code>
<code>template<class T> inline T const& get_optional_value_or (optional<T> const& _opt, T const& _default);</code>	<code>T const& optional<T>::get_value_or(T const& _default) const;</code>
<code>template<class T> inline T & get_optional_value_or(optional<T>& _opt, T& _default);</code>	<code>T& optional<T>::get_value_or(T& _default);</code>

VII-C-9 - Aide à la construction

- Aide à la construction :

```
template<class non_ref(T)> inline optional<T> make_optional(T const& valeur);
template<class non_ref(T)> inline optional<T> make_optional(bool
_b_initialise, T const& valeur );
```
- Objectif** : Construire une instance de *boost::optional*.
- Post-condition** : Retourne l'objet construit.
- Exception(s)** : Voir le constructeur correspondant.
- Notes** : La première méthode appelle le constructeur *optional<T>::optional(T const&)*.
La seconde méthode appelle le constructeur *optional<T>::optional (bool _b_initialise, T const&)*.

VII-C-10 - Opérateurs relationnels

- Opérateur d'égalité :

```
template<class T> inline bool operator== (optional<T> const& _l, optional<T> const& _r )
template<class T> inline bool operator== (optional<T> const& _l, T const& _r )
template<class T> inline bool operator== (T const& _l, optional<T> const& _r )
template<class T> inline bool operator== (optional<T> const& _l, boost::none_t )
template<class T> inline bool operator== (boost::none_t _l, optional<T> const& _r )
```

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

_l	_r	Résultat
Non initialisé	Non initialisé	true
Non initialisé	Initialisé	false
Initialisé	Non initialisé	false
Initialisé	Initialisé	comparaison des valeurs : (*_l) == (*_r)

- Exception(s)** : Aucune.
- Notes 1** : Si les deux éléments sont initialisés, la comparaison se fait directement sur les objets contenus.

- **Notes 2** : L'opérateur a été surchargé pour toutes les combinaisons entre *boost::optional*, une variable du type *T* et *boost::none*. Notons cependant l'absence d'une surcharge entre deux *boost::optional* de types différents (*boost::optional<T>==boost::optional<U>*) à l'instar des constructeurs existants.

Opérateur de différence :

```
template<class T> inline bool operator!= (optional<T> const& _l, optional<T> const& _r )
template<class T> inline bool operator!= (optional<T> const& _l, T const& _r )
template<class T> inline bool operator!= (T const& _l, optional<T> const& _r )
template<class T> inline bool operator!= (optional<T> const& _l, boost::none_t _r )
template<class T> inline bool operator!= (boost::none_t _l, optional<T> const& _r )
```

- **Objectif** : Surcharge de l'opérateur de différence pour deux *boost::optional*.
- **Notes** : La surcharge est strictement équivalente à *!(_l==_r)*.

Opérateur inférieur strict :

```
template<class T> inline bool operator < (optional<T> const& _l, optional<T> const& _r )
template<class T> inline bool operator < (optional<T> const& _l, T const& _r )
template<class T> inline bool operator < (T const& _l, optional<T> const& _r )
template<class T> inline bool operator < (optional<T> const& _l, boost::none_t _r )
template<class T> inline bool operator < (boost::none_t _l, optional<T> const& _r )
```

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

_l	_r	Résultat
Non initialisé	Non initialisé	false
Non initialisé	Initialisé	true
Initialisé	Non initialisé	false
Initialisé	Initialisé	comparaison des valeurs : (*_l) < (*_r)

- **Exception(s)** : Aucune.
- **Notes 1** : On considère que l'état non initialisé est le plus petit.
- **Notes 2** : La surcharge de l'opérateur '<' permet d'utiliser *boost::optional* avec les *std::map*.

Opérateur supérieur strict :

```
template<class T> inline bool operator > (optional<T> const& _l, optional<T> const& _r )
template<class T> inline bool operator > (optional<T> const& _l, T const& _r )
template<class T> inline bool operator > (T const& _l, optional<T> const& _r )
template<class T> inline bool operator > (optional<T> const& _l, boost::none_t _r )
template<class T> inline bool operator > (boost::none_t _l, optional<T> const& _r )
```

- **Objectif** : Surcharge de l'opérateur '>'.
- **Notes** : Strictement équivalent à *_r<_l*

Opérateur inférieur ou égal :

```
template<class T> inline bool operator <= (optional<T> const& _l, optional<T> const& _r )
template<class T> inline bool operator <= (optional<T> const& _l, T const& _r )
template<class T> inline bool operator <= (T const& _l, optional<T> const& _r )
template<class T> inline bool operator <= (optional<T> const& _l, boost::none_t _r )
template<class T> inline bool operator <= (boost::none_t _l, optional<T> const& _r )
```

- **Objectif** : Surcharge de l'opérateur '<='.
- **Notes** : Strictement équivalent à *!(_r<_l)*

Opérateur supérieur ou égal :

```
template<class T> inline bool operator >= (optional<T> const& _l, optional<T> const& _r )
template<class T> inline bool operator >= (optional<T> const& _l, T const& _r )
template<class T> inline bool operator >= (T const& _l, optional<T> const& _r )
template<class T> inline bool operator >= (optional<T> const& _l, boost::none_t _r )
template<class T> inline bool operator >= (boost::none_t _l, optional<T> const& _r )
```

- **Objectif** : Surcharge de l'opérateur '>='.
- **Notes** : Strictement équivalent à *!(_l<_r)*

VII-C-11 - Fonction d'échange : swap

Surcharge de swap :

```
template<class T> inline void swap (optional<T>& _l, optional<T>& _r )
```

- **Objectif** : Surcharge de la méthode *swap*.
- **Post-condition** :

_l	_r	Résultat
Non initialisé	Non initialisé	Les deux paramètres restent inchangés.
Non initialisé	Initialisé	<code>_l = _r;</code> <code>_r = boost::none;</code>
Initialisé	Non initialisé	<code>_r = _l;</code> <code>_l = boost::none;</code>
Initialisé	Initialisé	<code>swap(*_l, *_r);</code>

- **Exception(s)** : Celle(s) levée(s) par swap si les deux objets sont initialisés, ou celle(s) des opérateurs d'affectation si un objet est initialisé mais pas l'autre.
- **Notes** : La méthode *swap* est utilisée non qualifiée mais un `using std::swap` précède cette utilisation.

VII-D - Interaction avec les flux

Des surcharges sont définies pour les opérateurs d'entrée et de sortie vers les flux adéquats. Ces surcharges ne sont pas documentées, donc ne sont peut être pas pérennes. Je pense qu'elles se conçoivent plus dans une perspective de test (dump) que pour une sérialisation.

Le fichier d'en-tête est :

```
boost/optional/optional_io.hpp
```

Les deux opérateurs surchargés sont décrits ci-dessous.

Ecriture d'un `boost::optional` :

```
template<class CharType, class CharTrait, class T>
inline std::basic_ostream<CharType, CharTrait>&
operator<<(std::basic_ostream<CharType, CharTrait>& _flux, optional<T> const& _opt)
```


- **Objectif** : Ecriture d'un `boost::optional`.
- **Post-condition** : Si `_opt` n'est pas initialisé, alors c'est équivalent à `_flux <<"--"`, sinon c'est équivalent à `_flux <<' '<<*_opt`.
- **Exception(s)** : Aucune.

Lecture d'un `boost::optional` :


```
template<class CharType, class CharTrait, class T>
inline std::basic_istream<CharType, CharTrait>&
operator>>(std::basic_istream<CharType, CharTrait>& _flux, optional<T>& _opt)
```

- **Objectif** : Lecture d'un `boost::optional`.
- **Post-condition** : Si le flux débute par un espace (' ') alors un `T` est lu et affecté à `opt`, sinon `opt` est positionné à non initialisé.
- **Exception(s)** : Aucune.



VII-E - Limitations et contraintes

- Si la bibliothèque Boost.Optional est utilisée conjointement avec  **Boost.In Place Factory**, alors aucune exigence n'est imposée aux constructeurs de `T`.

Sinon, T doit être constructible par copie.

- Il n'y a pas d'exigence pour que T soit constructible par défaut.
- T ne doit pas générer d'exception dans son destructeur. Le cas échéant, les méthodes faisant appel au destructeur ne sont pas sûres vis à vis des exceptions. Une valeur indéterminée peut résulter de la levée d'une exception.
- *optional<bool>* ne doit pas être confondu avec un booléen à trois états. Outre que les opérateurs relationnels ne sont pas conçus pour être cohérents avec ce type de logique, il faut garder en mémoire que l'état non-initialisé correspond à une valeur indéterminée. Donc, son utilisation en tant que valeur est une erreur. Pour un booléen trivalent, Boost propose la bibliothèque  **Boost.Tribool**.
- Contrairement à ce que pourraient laisser entendre les surcharges des opérateurs '*' et '->', *boost::optional<T>* ne modélise pas un pointeur sur T, mais bien une enveloppe sur T. Les opérations de copie ou de test sur un pointeur se font sur l'adresse. Ces mêmes opérations sur un *boost::optional* se font sur la valeur. Il en résulte que *boost::optional* ne suit pas la sémantique d'un pointeur.

Lorsque Boost.Optional est utilisé avec une référence : *boost::optional<T&>*, alors les fonctions suivantes ne sont pas disponibles :

- Constructeur avec conversion de type ;
- Constructeur avec  **Boost.In Place Factory** ;
- Opérateur d'assignement avec conversion de type ;
- Opérateur d'assignement avec  **Boost.In Place Factory**.

Contrairement à ce que dit la documentation officielle, l'accès par pointeur est disponible pour une instance avec référence (depuis la version Boost 1.34.0).

VII-F - En résumé

En-tête de la classe :	boost/optional.hpp boost/optional/optional.hpp
Espace de nommage :	boost
Définition d'une variable :	boost::optional<T> opt;
Valeur non initialisée :	boost::none (boost/none.hpp)
Accesseur interne par valeur :	get, get_value_or, operator*
Accesseur externe par valeur :	get, get_optional_value_or
Accesseur interne par adresse :	get_ptr, get_value_or, operator->
Accesseur externe par adresse :	get, get_pointer
Tester l'état :	if(opt) ou if(!opt)

VIII - Comment ça marche ?

L'implémentation est longue mais pas trop compliquée. Elle ne va pas être reprise dans son intégralité ici car ce serait de peu d'intérêt. J'encourage vivement le lecteur curieux et désireux de parfaire sa connaissance du C++ en général et de Boost en particulier de s'attarder à décortiquer cette implémentation.

Seuls quelques points vont être abordés, soit parce qu'ils présentent des subtilités, soit parce qu'ils sont intéressants par l'utilisation d'une librairie Boost tierce, soit tout simplement parce qu'ils présentent un caractère formateur vis-à-vis du langage.

Les détails présentés ci-dessous n'ont bien sûr pas à être utilisés dans le code utilisateur car ils relèvent de l'implémentation.

Le code présenté ici n'est pas stricto sensu le code des bibliothèques Boost. Il est réécrit de façon 'allégée' pour mieux faire comprendre son fonctionnement ou son principe. Ceci peut faire perdre à certaines classes une partie de leur définition et altérer leur sémantique. J'ai jugé que ces éléments n'étaient pas forcément pertinents au moment de l'explication. Que le lecteur avisé me pardonne et comprenne ma démarche.

VIII-A - Zone de stockage de la valeur

La valeur doit être stockée sur un emplacement mémoire correctement aligné. Une classe spécifique est utilisée :

Classe conteneur :

```
template <class T>
class aligned_storage
{
    union dummy_u
    {
        char data[ sizeof(T) ];
        typename type_with_alignment<
            ::boost::alignment_of<T>::value >::type aligner_;
    } dummy_ ;

public:

    void const* address() const { return &dummy_.data[0]; }
    void * address() { return &dummy_.data[0]; }
};
```

type_with_alignment et *alignment_of* sont issues de Boost.Type Traits. Elles permettent d'assurer un alignement correct du buffer destiné à contenir la valeur, c'est à dire l'instance de la classe T pour un *optional<T>* ou la référence pour un *optional<T&>*.

Mais, ce n'est pas fini ! Une classe enveloppe va être utilisée pour les références :

Un wrapper pour les références

```
template <typename RefT>
class reference_content
{
private:
    RefT content_;

public:
    ~reference_content()
    {
    }

    reference_content(RefT r)
        : content_( r )
    {
    }

    reference_content(const reference_content& operand)
        : content_( operand.content_ )
    {
    }

private:
    reference_content& operator=(const reference_content&);

public:
    RefT get() const
    {
        return content_;
    }
};
```

Ensuite, une méta-fonction est définie pour utiliser cette classe à bon escient :

Méta-fonction :

```
template <typename T>
struct make_reference_content
{
    typedef T type;
};

template <typename T>
struct make_reference_content< T& >
{
    typedef reference_content<T&> type;
};
```

La méta-fonction *make_reference_content* 'retourne' le type T si elle est 'appelée' sur un type en tant que tel, et retourne l'enveloppe *reference_content<T&>* lorsqu'elle est 'appelée' avec une référence.

Ces éléments sont combinés pour déclarer l'espace de stockage :

class optional_base : emplacement pour la valeur

```
template<class T>
class optional_base
{
    ...
    typedef ::boost::detail::make_reference_content<T>::type internal_type ;
    typedef aligned_storage<internal_type> storage_type ;
    ...
    storage_type m_storage ;
};
```

Jouons au compilateur : pour *boost::optional<T>*, *internal_type* s'évalue à T, retour de la méta-fonction *make_reference_content<T>*. Donc *storage_type* est évalué à *aligned_storage<T>*. Notre espace de stockage est (modulo l'alignement) : *char data[sizeof(T)]*.

Maintenant, avec une référence : *boost::optional<T&>*, la méta-fonction utilisée devient *make_reference_content<T&>*, ce qui donne pour *internal_type* : *reference_content<T&>*. *storage_type* est évalué à *aligned_storage<reference_content<T&>>*. Notre espace de stockage est (modulo l'alignement) : *char data[sizeof(reference_content<T&>)]*.

Notons que la valeur est toujours stockée dans *aligned_storage::data*, le type ne sert que pour la taille.

Pourquoi passer par une enveloppe pour la référence ? La réponse est dans le standard :

Standard 5.3.3 Sizeof (2) :

When applied to a reference or a reference type, the result is the size of the referenced type.

Pour vous en convaincre, essayez simplement ce code :

```
#include <iostream>
template<class T>
struct mon_enveloppe
{
    T m_t;
};

int main()
{
    std::cout<<"Taille enveloppe : "
        <<sizeof(mon_enveloppe<char>)
        <<" / Taille type "
        <<sizeof(char)
        <<std::endl;
    std::cout<<"Taille enveloppe& : "
        <<sizeof(mon_enveloppe<char&>)
        <<" / Taille type& "
        <<sizeof(char&);
}
```

```
<<std::endl;
return 0;
}
```

VIII-B - Gérer les références

Deux classes traits vont permettre de gérer les types selon que *boost::optional* est instancié avec une référence ou non :

```
template<class T>
struct types_when_isnt_ref
{
    typedef T const& reference_const_type ;
    typedef T & reference_type ;
    typedef T const* pointer_const_type ;
    typedef T * pointer_type ;
    typedef T const& argument_type ;
} ;

template<class T>
struct types_when_is_ref
{
    typedef typename remove_reference<T>::type raw_type ;

    typedef raw_type& reference_const_type ;
    typedef raw_type& reference_type ;
    typedef raw_type* pointer_const_type ;
    typedef raw_type* pointer_type ;
    typedef raw_type& argument_type ;
} ;
```

remove_reference est une méta-fonction issue de Boost.Type Traits. Elle retourne le type sans la référence. Cette méta-fonction est équivalente à :

```
template<class T>
struct remove_reference
{
    typedef T type;
};

template<class T>
struct remove_reference<T&>
{
    typedef T type;
};
```

La détermination des types dans la classe *boost::optional* (en fait *boost::optional_base*) passe après plusieurs étapes par l'utilisation de Boost.MPL. Détaillons ces étapes !

```
typedef types_when_isnt_ref<T> types_when_not_ref ;
typedef types_when_is_ref<T> types_when_ref ;
```

Deux petits synonymes pour faciliter la lecture.

```
typedef typename is_reference<T>::type is_reference_predicate ;
```

is_reference est une méta-fonction de la librairie Boost.Type Traits. Cette méta-fonction est équivalente à :

```
template< typename T > struct is_reference
{
    static const int value =false;
};

template< typename T > struct is_reference< T& >
{
```

```
static const int value =true;
};
```

Le tout va être utilisé comme suit :

```
typedef typename mpl::if_<is_reference_predicate,types_when_ref,types_when_not_ref>::type types ;
```

`mpl::if_c` est une méta-fonction de la bibliothèque MPL dont la définition est semblable à :

```
template<bool C, typename TVrai, typename TFaux>
struct mpl::if_
{
    typedef TVrai type;
};

template<typename TVrai, typename TFaux>
struct mpl::if_<false, TVrai, TFaux>
{
    typedef TFaux type;
};
```

La méta-fonction prend la valeur *TVrai* si *C* s'évalue à *true* et à *TFaux* si *C* s'évalue à *false*.

La méta-fonction est ici appelée avec le prédicat *is_reference_predicate*, qui, rappelons-le, est une méta-fonction évaluant à *true* pour une référence et à *false* sinon. Les deux classes traits définies ci-dessus forment le résultat !

Finalement, les types souhaités sont ramenés simplement dans la classe :

```
typedef typename types::reference_type      reference_type ;
typedef typename types::reference_const_type reference_const_type ;
typedef typename types::pointer_type       pointer_type ;
typedef typename types::pointer_const_type pointer_const_type ;
typedef typename types::argument_type      argument_type ;
```

Décortiquons le tout pour nos deux instances possibles.

`boost::optional<T>` :

- 1 `is_reference<T>` s'évalue à *false* ;
- 2 `mpl::if_<false,types_when_ref,types_when_not_ref>` s'évalue à *types_when_not_ref*, soit *types_when_isnt_ref<T>*
- 3 On a donc :

```
template<class T>
class optional_base
{
    ...
    typedef T const& reference_const_type ;
    typedef T &      reference_type ;
    typedef T const* pointer_const_type ;
    typedef T *      pointer_type ;
    typedef T const& argument_type ;
    ...
}
```

`boost::optional<T&>` :

- 1 `is_reference<T&>` s'évalue à *true* ;
- 2 `mpl::if_<true,types_when_ref,types_when_not_ref>` s'évalue à *types_when_ref*, soit *types_when_is_ref<T&>*
- 3 On a donc :

```
template<class T>
class optional_base
```

```
{
...
typedef T& reference_const_type ;
typedef T& reference_type ;
typedef T* pointer_const_type ;
typedef T* pointer_type ;
typedef T& argument_type ;
...
}
```

Tout cela constitue un exemple intéressant de l'utilisation de la méta-programmation !

VIII-C - Vérifier l'état

L'explication s'inspire de l'article suivant :  **The Safe Bool Idiom**, de Bjorn Karlsson.

L'objectif est de pouvoir tester l'état d'une instance de *boost::optional* avec la même facilité qu'un booléen. *boost::optional* propose 2 alternatives pour tester l'état d'une instance :

- `operator unspecified-bool-type() const ;`
- `bool optional<T>::operator!() const;`

Ecartons tout de suite la seconde forme qui est tout simplement implémentée par :

```
bool operator!() const { return !this->is_initialized() ; }
```

is_initialized est un accesseur sur la variable membre *m_initialized* valant *true* si l'instance est initialisée, et *false* sinon. Aucune subtilité dans ce code. Notons toutefois que l'état initialisé peut être testé de facto par l'utilisation de `!!` :

```
if (!!opt) { ... }
```

Revenons donc à la première méthode. Pourquoi ne pas avoir simplement redéfini un opérateur de trans-typage vers `bool` ? Pourquoi le code suivant n'est-il pas satisfaisant ?

```
operator bool() const { return this->is_initialized(); }
```

Plusieurs raisons viennent invalider cette solution :

- `bool` est un type de base et toute conversion depuis `bool` vers d'autres types de base est alors implicite :

```
boost::optional<MaClasse> opt;
double valeur = opt; // code qui serait valide !
```
- toujours dans la même veine, étant un type de base, des opérations comme l'addition ou le décalage de bits sont rendues possibles :

```
boost::optional<MaClasse> opt;
opt << 12; // code qui serait valide !
```
- si une autre classe possède aussi une surcharge de transtypage vers `bool`, des comparaisons entre deux instances de ces classes différentes deviennent valides :

```
class B{
...
operator bool() const;
};
boost::optional<MaClasse> opt;
B mon_b;
if (opt==b) { // code qui serait valide ! }
```

Toutes ces opérations rendues valides n'ont cependant aucun sens pour la classe *boost::optional*. Il faut les interdire. Le transtypage vers *bool* n'est donc pas une solution.

Le passage par un pointeur vers une méthode membre permet de résoudre ces différents problèmes. L'implémentation suit alors ce schéma :

```
template<class T>
class optional_base
{...
    typedef optional_base<T> this_type ;
    ...
    typedef bool (this_type::*unspecified_bool_type) () const;
    ...
    unspecified_bool_type safe_bool() const
    { return m_initialized ? &this_type::is_initialized : 0 ; }
    ...
    // la surcharge qui nous intéresse :
    operator unspecified_bool_type() const
    { return this->safe_bool() ; }
```

Les erreurs précédentes sont rendues invalides. Cependant, comme mentionné dans l'article, cette solution va de pair avec la surcharge des opérateurs relationnels `operator==` et `operator!=`. En effet, en l'absence de cette surcharge, la comparaison de deux instances est valide mais sa sémantique est erronée :

```
class A
{
...
    typedef A this_type ;
    typedef bool (this_type::*unspecified_bool_type) () const;
...
public:
...
    operator unspecified_bool_type() const
    { return m_initialized ? &this_type::is_initialized : 0 ; }
...
private:
    bool m_initialized;

    bool is_initialized() const { return m_initialized; }
};

int main()
{
    A a;
    A a2;
    if(a==a2){
        std::cout<<"mauvaise semantique !"<<std::endl;
    }
...
    return 0;
}
```

VIII-D - Opérateur d'affectation

VIII-D-1 - Déterminer la bonne surcharge

L'opérateur d'affectation (pour la surcharge avec un élément) est défini comme suit :

```
optional& operator= ( argument_type val )
{
    this->assign( val ) ;
    return *this ;
}
```

Regardons à quoi ressemble cette méthode `assign` :

```
void assign ( argument_type val )
```

```
{
    if (is_initialized())
        assign_value(val, is_reference_predicate() );
    else construct(val);
}
```

`assign_value` est surchargée sur les deux formes suivantes :

```
void assign_value ( argument_type val, is_not_reference_tag );
void assign_value ( argument_type val, is_reference_tag);
```

Revenons sur notre méta-fonction `is_reference_predicate`. Présentée brièvement ci-dessus, détaillons là un peu plus :

```
typedef is_reference<T>::type is_reference_predicate ;
template< typename T > struct is_reference
    : ::boost::integral_constant<bool,false>
{ };
template< typename T > struct is_reference< T& >
    : ::boost::integral_constant<bool,true>
{ };

struct integral_constant : public mpl::integral_c<T, val>
{
    typedef integral_constant<T,val> type;
};
```

`is_reference` est une méta-fonction avec une première définition générique et une spécialisation lorsque le paramètre est une référence. `is_reference_predicate` 'est un' `::boost::integral_constant<bool,false>` pour un type et 'est un' `::boost::integral_constant<bool,true>` pour une référence : c'est le résultat de `integral_constant::type` qui renvoi sur lui-même. Enfin, `::boost::integral_constant`, est définie comme suit :

```
struct integral_constant : public mpl::integral_c<T, val>
{
    typedef integral_constant<T,val> type;
};

template<> struct integral_constant<bool,true> : public mpl::true_
{
    typedef integral_constant<bool,true> type;
};
template<> struct integral_constant<bool,false> : public mpl::false_
{
    typedef integral_constant<bool,false> type;
};
```

Les deux dernières spécialisations montrent que `is_reference_predicate` 'est un' `mpl::true_` pour `boost::optional<T&>`; et `is_reference_predicate` 'est un' `mpl::false_` pour `boost::optional<T>`.

Puisque d'autre part, on a :

```
typedef mpl::true_ is_reference_tag ;
typedef mpl::false_ is_not_reference_tag ;
```

on en déduit que la surcharge de `assign` appelée est :

- `boost::optional<T> => assign_value(val, is_not_reference_tag());`
- `boost::optional<T&> => assign_value(val, is_reference_tag());`

VIII-D-2 - Réassigner une référence

Rappelons les deux formes que prend `assign_value` :

```
void assign_value ( argument_type val, is_not_reference_tag ) { get_impl() = val; }  
void assign_value ( argument_type val, is_reference_tag      ) { construct(val); }
```

Lorsque le type n'est pas une référence, *get_impl* retourne le conteneur interne correctement typé, donc c'est l'affectation pour ce type qui entre en jeu.

Lorsque le type est une référence, alors la méthode *construct* est utilisée. Effectivement, en C++, on ne sait pas réassigner une référence par affectation. Donc il faut réassigner la valeur de la même façon que lorsqu'on la construit. Cette méthode est simplement définie comme suit :

```
void construct ( argument_type val )  
{  
    new (m_storage.address()) internal_type(val) ;  
    m_initialized = true ;  
}
```

La définition est la même que le type soit une référence ou non. Le placement new permet d'appeler le constructeur par copie de *internal_type*. Comme vu plus haut, *internal_type* vaut T pour *boost::optional<T>* et *reference_content<T&>* pour *boost::optional<T&>*. L'enveloppe permet de réassigner la référence grâce à son constructeur d'initialisation :

```
reference_content (T& r)  
    : content_( r )  
{ }
```