

Mariage de la Programmation Orientée Objet et de la Programmation Générique : Type Erasure

par Alp Mestan ([Site personnel](#)) ([Blog](#))

Date de publication :

Dernière mise à jour :

Si vous utilisez les templates du C++ pour écrire des composants génériques, vous avez déjà peut-être voulu les combiner à la programmation orientée objet, via de l'héritage par exemple. Peut-être avez-vous eu des difficultés ; cet article va vous présenter une technique répandue, nommée *Type Erasure*, qui vous permettra de tirer profit des deux mondes sans perdre en flexibilité ni en maintenabilité.

Commentez

Vous pouvez trouver les codes C++ associés aux parties III, IV et V de cet article dans l'archive ZIP suivante : [Source type_erasure.zip](#).

| | |
|---|----|
| I - Introduction..... | 3 |
| II - Etude de cas..... | 3 |
| III - Principe de Type Erasure..... | 4 |
| IV - Application dans un cas concret..... | 5 |
| V - Réécrivons boost::any..... | 10 |
| VI - Limites et conclusion..... | 14 |
| VII - Remerciements..... | 14 |

I - Introduction

Le C++ est un langage riche. On peut résoudre des problèmes identiques avec des approches différentes. Certains utiliseront des classes, d'autres un style plus basé sur les fonctions, etc. En effet, le C++ supporte plusieurs paradigmes, comme la programmation orientée objet, la programmation générique, la programmation fonctionnelle, la programmation logique, etc. Toutefois, seuls les deux premiers ici sont nativement supportés par le C++ (c'est à dire sans utiliser de bibliothèque). Seulement, il peut arriver que l'on ait besoin de mélanger les styles, afin de bénéficier des avantages de l'un et de l'autre et l'on aimerait bien ne pas avoir de difficultés à les mélanger. En effet, lorsque l'on écrit des composants génériques (paramétrés via le mécanisme de templates), on veut pouvoir les injecter dans du code Orienté Objet ne serait-ce par exemple qu'en agissant uniformément sur une liste de composants via le polymorphisme d'héritage... Or ceci n'est pas trivial et possède ses pièges.

Ce n'est toutefois pas si élémentaire de tirer profit, dans notre cas, du mélange de la programmation orientée objet avec la programmation générique, de par les règles qui régissent le langage C++. Par exemple, la plupart d'entre nous ont probablement déjà fait face au problème suivant : soit une classe template définie comme suit.

```
template <class T>
class A
{
    T t;
    // ...
};
```

Comment peut-on alors stocker ensemble des objets A<T1>, A<T2>, etc, où T1, T2 et autres sont des types différents ? C'est l'une des questions qui va être au centre de cet article. Nous allons dans un premier temps mettre en avant le problème auquel répond le principe de *Type Erasure*, puis nous introduisons ce dernier en donnant une solution à notre exemple de l'étude de cas. Enfin, nous appliquerons le principe dans un exemple plus concret et utile, puis nous réécrivons la classe boost::any avant de conclure cet article sur les limites de cette approche.

II - Etude de cas

Nous allons commencer par un problème trivial : prenons la classe template suivante.

```
template <typename T>
class A
{
    T t;
public:
    A(const T& t_) : t(t_) { }
    void print(std::ostream& o) const { o << t; }
};
```

A première vue, rien de très compliqué. Une classe template qui renferme une valeur d'un certain type, et qui permet d'envoyer cette valeur dans un flux. Elle n'est pas terriblement utile, mais sera une excellente base.

Désormais, imaginons vouloir stocker un nombre arbitraire d'objets de ce type, avec des paramètres *T* différents, dans un même conteneur. Ainsi, nous voudrions avoir par exemple un `std::vector` contenant des A<int>, A<std::string>, A<char>, etc.

Contrairement à ce que l'on peut penser parfois, le type A<int> est différent de A<char>, et on ne peut pas faire un `std::vector<A>` ou autre `std::vector< A<int> >` qui permettrait de stocker des A<T> quels que soient les types T. N'oubliez pas en effet que écrire A<int> génère une classe A<int> en remplaçant T par int dans le code de la classe. A<char> et A<int> n'ont en commun que le code qui a permis de les générer, mais ce sont deux types incompatibles.

Toutefois, le fait qu'elles proviennent d'un même "modèle", "patron", va nous servir ici. Voyons maintenant la solution qui utilise, vous vous en doutez étant donné le titre de l'article, l'héritage.

III - Principe de Type Erasure

La clé ici est que nous voulons d'une part avoir des `A<T>` à manipuler en tant que tels, et d'autre part nous voulons pouvoir stocker des `A<>` avec des types `T` différents en manipulant l'ensemble de manière uniforme. Dans notre cas, nous voulons stocker les `A<T>` pour pouvoir les afficher à travers un flux `std::ostream`. Vous vous en doutez, c'est une mission pour l'héritage et la virtualité. Écrivons donc une classe abstraite qu'implémenteront toutes les classes `A<T>` quel que soit le type `T`.

```
class A_base
{
    virtual void do_print(std::ostream&) const = 0;
public:
    void print(std::ostream& o) const { do_print(o); }
};
```

Il nous suffit ensuite de modifier notre classe template `A<T>` pour qu'elle hérite et implémente cette classe abstraite.

```
template <typename T>
class A : A_base
{
    T t;
    void do_print(std::ostream& o) const { o << t; }
public:
    A(const T& t_) : t(t_) { }
};
```

Et voilà, nous pouvons désormais d'une part stocker tous les `A<T>` dans un même conteneur via la classe abstraite `A_base`, et d'un autre côté préserver leur utilisation sans passer par l'interface `A_base`. Le coût n'est absolument pas grand. Il y aura simplement une *vtable* contenant toute l'interface publique par laquelle on manipulera uniformément les différents `A<T>`. Il y a un deuxième coût toutefois : vous serez obligé de stocker des pointeurs au lieu de stocker des valeurs simples. Voici donc notre `std::vector` et une manipulation uniforme sur les `A<T>`, dans un code complet que vous pourrez compiler si vous avez Boost. Sinon, remplacez les pointeurs intelligents par des `A_base*` et faites des `delete` à la fin de main sur ces `A_base*`.

```
#include <iostream>
#include <string>
#include <vector>
#include <tr1/shared_ptr.h>
// si votre compilateur ne fournit pas tr1/shared_ptr.h
// vous pouvez obtenir la même chose depuis
// <boost/tr1/memory.hpp>
// qui est donc inclus dans Boost

class A_base
{
    virtual void do_print(std::ostream&) const = 0;
public:
    void print(std::ostream& o) const { do_print(o); }
};

template <typename T>
class A : public A_base
{
    T t;
    void do_print(std::ostream& o) const { o << t; }
public:
    A(const T& t_) : t(t_) { }
};
```

```

// permet d'avoir des A< A<T> >
std::ostream& operator<<(std::ostream& o, const A_base& a)
{
    a.print(o);
    return o;
}

int main()
{
    std::vector< std::tr1::shared_ptr<A_base> >
    vec; // on utilise std::tr1::shared_ptr, un des pointeurs intelligents de Boost/TR1, cf [1] [2]
    vec.push_back(std::tr1::shared_ptr<A_base>(new A<int>(42)));
    vec.push_back(std::tr1::shared_ptr<A_base>(new A<std::string>("Type Erasure")));
    vec.push_back(std::tr1::shared_ptr<A_base>(new A<char>('c')));
    std::vector< std::tr1::shared_ptr<A_base> >::iterator it = vec.begin(); // vivement auto de C++0x ;- )
    for ( ; it != vec.end(); it++ )
    {
        (*it)->print(std::cout);
    }
    return 0;
}

```

[1] [Pointeurs Intelligents](#), Loïc Joly

[2] [Boost.SmartPtr, les pointeurs intelligents de Boost](#), par Matthieu Brucher

Ce code affiche donc

42Type Erasurec.

Certes, vous voyez que ce code marche très bien dans notre cas simple, mais vous vous demandez quelle utilité il peut avoir dans la pratique, dans nos projets de tous les jours. C'est pourquoi dans la section suivante nous allons nous attaquer à un problème plus sérieux et plus réaliste.

IV - Application dans un cas concret

En C++ moderne, l'une des bonnes pratiques lorsque l'on écrit des composants modulaires (souples, paramétrables) est de découper ses composants en classes de politiques (*Policy Classes*) comme décrit dans l'article [Classes de Traits et de Politiques](#) ou encore dans le livre *Modern C++ Design* de Andrei Alexandrescu, évangéliste de cette pratique.

C'est pourquoi nous allons avoir affaire à un système de génération de widgets paramétré par des politiques. Celui-ci sera minimal pour garder le fil de l'article, mais tout de même consistant.

```

#include <iostream>
#include <string>

#define CHECK_FUN(Policy, Member) enum { Policy##Member = sizeof(&Policy::Member) > 0 };

template
<
    class SizePolicy,
    class ClickAwarenessPolicy,
    class TextPolicy,
    class DrawingPolicy
>
class widget_impl :
public SizePolicy,
public ClickAwarenessPolicy,
public TextPolicy,
public DrawingPolicy
{
    /* Par sûreté, on vérifie la présence des fonctions nécessaires
    dans les politiques */
    CHECK_FUN(DrawingPolicy, draw)
}

```

```

public:
widget_impl(unsigned int width = 800, unsigned int height = 600, const std::string& text = "")
    :SizePolicy(width, height), TextPolicy(text)
{
}

void print_size() const
{
    std::cout <<SizePolicy::width << "x" <<SizePolicy::height << std::endl;
}
};

struct NoText { };
struct NonClickAware { };

struct NonResizable
{
protected:
    unsigned int width;
    unsigned int height;

public:
    NonResizable(unsigned int w, unsigned int h) : width(w), height(h) { }
};

struct Resizable
{
protected:
    unsigned int width;
    unsigned int height;

public:
    Resizable(unsigned int w, unsigned int h) : width(w), height(h) { }

void resize(unsigned int w, unsigned int h)
{
    width = w;
    height = h;
}
};

struct ClickAware
{
    void on_click()
    {
        std::cout << "Clicked" << std::endl;
    }
};

struct ReadOnlyText
{
protected:
    std::string text;

public:
    ReadOnlyText(const std::string& t) : text(t) { }
    std::string get_text() { return text; }
};

struct EditableText
{
protected:
    std::string text;

public:
    EditableText(const std::string& t) : text(t) { }
    std::string get_text() { return text; }
    void set_text(const std::string& s) { text = s; }
};

struct RectangularDraw

```

```

{
    void draw()
    {
        std::cout << "      " << std::endl;
        std::cout << "|      |" << std::endl;
        std::cout << "|      |" << std::endl;
    }
};

struct SquareDraw
{
    void draw()
    {
        std::cout << "      " << std::endl;
        std::cout << "|      |" << std::endl;
        std::cout << "|      |" << std::endl;
        std::cout << "|      |" << std::endl;
    }
};

int main()
{
    typedef widget_impl <
        NonResizable,
        NonClickAware,
        ReadOnlyText,
        SquareDraw
        > widget_t;

    widget_t w(1024, 780, "The best widget of the world");
    w.draw();
    std::cout << "Widget's text : " << w.get_text() << std::endl;
    w.print_size();
    // w.set_text("foo"); ne compile pas, car ReadOnlyText
    // w.resize(); ne compile pas, car NonResizable
    // w.on_click(); ne compile pas, car NonClickAware

    return 0;
}


```

Résumons... Nous avons une classe template `widget_impl`, paramétrée par différentes politiques :

- Politique de redimensionnement (Resizable ou NonResizable),
- Politique de réaction aux clics (ClickAware ou NonClickAware),
- Politique textuelle (Aucun texte inclus, Texte inclus en lecture, Texte inclus et modifiable),
- Politique de dessin (Dessin de carré, Dessin de rectangle).

Nous avons ensuite défini des implémentations de ces politiques. Puis dans la fonction `main`, nous définissons un certain type de widget `widget_t` qui n'est pas redimensionnable, ne réagit pas aux clics, affiche du texte sans possibilité de l'éditer et dont le dessin représente un carré. Déjà, quel intérêt n'est-ce pas ? Tout simplement, nous avons décomposé notre widget en fonctionnalités orthogonales (i.e. indépendantes) et pouvons ainsi soit utiliser des implémentations de fonctionnalités déjà définies, ou bien définir notre propre implémentation d'une politique donnée ! Par exemple, nous pourrions définir une implémentation de politique s'occupant de définir une fonction `draw` pour afficher un widget circulaire et passer cette implémentation de politique, sans avoir à hériter de notre type existant ou à redéfinir toute la classe widget, ou autres (comme c'est le cas avec beaucoup de toolkits GUI).

Maintenant que nous avons démontré l'utilité d'un tel design, il est temps de mettre le doigt sur un problème... fort embêtant.

 *Dans beaucoup de bibliothèques GUI, il y a un système de parent/enfants entre widgets. En effet, il n'est pas rare qu'une fenêtre ait la responsabilité de détruire tous les composants qu'elle comporte. Le mécanisme utilisé pour ce faire est de passer le composant parent au constructeur du composant enfant. Le composant enfant peut ainsi demander à son parent de le rajouter dans sa liste des widgets sous sa responsabilité. Mais comment faire pour qu'un type de widget donné, comme `widget_t`*

ci-dessus (qui correspond, je le rappelle, à widget_impl<NonResizable, NonClickAware, ReadOnlyText, SquareDraw> , puisse stocker des widget_impl<SomeSizePolicy, SomeClickAwarenessPolicy, SomeTextPolicy, SomeDrawingPolicy> , où chacune des implémentations de politique peut être différente d'un objet à l'autre qui doit être stocké dans la liste ?

Nous allons, comme dans la section précédente, introduire une classe de base permettant de profiter du polymorphisme d'héritage. Ainsi, dans l'interface (publique) de la classe de base, nous y définirons les fonctions virtuelles, éventuellement pures, qu'il nous faut pour que chaque widget puisse être sereinement responsable d'une liste de widgets enfants et puisse gérer leur durée de vie correctement. Nous allons d'abord définir un foncteur utilitaire, *deleter*.

```
// foncteur utilitaire qui permet de détruire en série tous les widgets enfants via std::for_each
struct deleter
{
    template <typename T>
    void operator() (T* t)
    {
        t->destroy();
    }
};
```

Puis définissons la classe de base, *widget*.

```
class widget
{
    std::list<widget*> children; // liste des widgets enfant, dont le widget courant est responsable
    widget* parent; // widget parent

    void register_child(widget* w)
    {
        children.push_back(w); // enregistre un widget enfant
    }

    widget(const widget& other); // widget est ainsi non-copiable
    widget& operator=(const widget& other); // ni assignable

public:
    widget(widget* parent_) : parent(parent_)
    {
        if(parent_ != NULL)
        {
            parent_->register_child(this); // si on donne un parent, alors c'est le parent qui devient responsable du widget
        }
    }

    virtual ~widget()
    {
        if(parent == NULL)
            widget::destroy(); // s'il n'y a pas de parent, le destructeur détruit le widget courant
            // sinon, c'est le parent qui s'en charge via la fonction destroy
            // cela permet d'éviter un double-appel à destroy
    }

    virtual void destroy()
    {
        std::for_each(children.begin(), children.end(), deleter());
        // appelle destroy sur tous les widgets enfants
    }
};
```

Biensûr, il faut désormais apporter de légères modifications à la classe template *widget_impl*. Tout d'abord, il faut la faire hériter de la classe *widget* et prendre cela en compte dans le constructeur de *widget_impl*.

```

template
<
    classSizePolicy,
    classClickAwarenessPolicy,
    classTextPolicy,
    classDrawingPolicy
>
class widget_impl :
    public widget, /* NOUVEAU */
    publicSizePolicy,
    publicClickAwarenessPolicy,
    publicTextPolicy,
    publicDrawingPolicy
{
    // ...
    public:
    widget_impl(widget* parent /* NOUVEAU */ , unsigned int width = 800, unsigned int
    height = 600, const std::string& text = "")
        : widget(parent) /* NOUVEAU */ , SizePolicy(width, height), TextPolicy(text)
        {
        }
    // ...
};
    
```

Il reste un dernier détail à régler. La fonction membre `widget::destroy` détruit les widgets enfants. Toutefois, imaginez que l'on ait des choses à détruire dans les politiques. Comment faire ? Il nous suffirait de procéder comme suit. Il faudrait définir une fonction membre `destroy` dans `widget_impl` qui aurait l'allure suivante.

```

template
<
    classSizePolicy,
    classClickAwarenessPolicy,
    classTextPolicy,
    classDrawingPolicy
>
class widget_impl :
    public widget, /* NOUVEAU */
    publicSizePolicy,
    publicClickAwarenessPolicy,
    publicTextPolicy,
    publicDrawingPolicy
{
    // ...
    public:
    // ...
    void destroy()
    {
        // ici on fait par exemple DrawingPolicy::destroy(); s'il y a quelque chose à détruire
        // et autres
        // enfin, on détruit les widget fils en déléguant le travail à widget::destroy
        widget::destroy();
    }
    // ...
};
    
```

Nous avons désormais appliqué le *Type Erasure* pour manipuler indifféremment des widgets basés sur une classe template. Si l'on voulait pouvoir gérer par exemple des redimensionnement en cascade, il faudrait rajouter le nécessaire (une fonction virtuelle pure `resize(int, int)` par exemple) dans l'interface publique de `widget`. En rajoutant une fonction qui permet de lister les widgets enfants dans l'interface de `widget`, ainsi qu'un peu d'affichage dans le constructeur et le destructeur de `widget_impl`, le code suivant permet de montrer que notre système marche très bien.

```

int main()
{
    typedef widget_impl <
        NonResizable,
    
```

```

NonClickAware,
ReadOnlyText,
SquareDraw
    > widget_t;

widget_t w(NULL, 1024, 780, "The best widget of the world");
widget_t w2(&w, 1024, 780, "The second best widget of the world");

w.print_children(); // fonction qui parcourt la liste des enfants et affiche le texte obtenu via get_text()
// ...
return 0;
}
    
```

Le code suivant affiche une construction, celle de `w` ; une autre, celle de `w2` ; puis cela affiche "The second best widget of the world" lors de l'appel à `print_children`. Et enfin, cela affiche deux destructions, celle de `w2` en premier, puis celle de `w`. Ainsi, `w2` ne se détruit pas tout seul mais laisse `w` s'en charger en appelant `destroy` sur `w2`.

J'ai donc ici montré l'utilité du principe de *Type Erasure* sur un exemple réel et concret. Le principe doit désormais être clair pour vous, et c'est pourquoi nous allons désormais passer à la création d'un outil qui existe déjà, mais dont vous n'aviez peut-être pas idée du fonctionnement avant : **`boost::any`**, une classe qui peut contenir des valeurs de type quelconque, et dont on peut changer la valeur et le type de valeur à l'exécution.

V - Réécrivons `boost::any`

Déjà, peut-être ne connaissez vous pas le module `Boost.Any` de `Boost`, qui définit la classe `boost::any`. Elle appartient depuis un bon nombre d'années à l'ensemble de bibliothèques `Boost`. Le problème résolu par *`boost::any`* est le suivant : je voudrais disposer d'un type qui puisse stocker des valeurs de (presque) n'importe quel autre type. La documentation, pour en découvrir plus, se situe [ici](#).

Regardons dans un premier temps comment s'utilise *`boost::any`*.

```

boost::any a(13); // a encapsule la valeur 13, de type int donc
a = std::string("salut"); // a encapsule une std::string contenant "salut"
    
```

Pour récupérer la valeur encapsulée par un *`boost::any`*, il existe une fonction (plusieurs en fait, mais du même nom), *`boost::any_cast`*. Exemple :

```

boost::any a(13);
int i = boost::any_cast<int>(a);
    
```

Si l'on utilise l'une des versions travaillant sur des références et que l'on donne un type incompatible (vers lequel on ne peut pas convertir la valeur encapsulée), alors une exception *`boost::bad_any_cast`* est lancée. Si l'on utilise l'une des versions travaillant sur des pointeurs et que l'on donne un type incompatible vers lequel convertir, *`boost::any_cast`* retourne un pointeur nul.

Voilà donc ce que nous allons ici reproduire, tout simplement, avec notre technique de *Type Erasure*.

Nous allons dans un premier temps créer une classe template qui permettra de stocker une valeur de n'importe quel type.

```

template <class T>
class value
{
    T t;

public:
    value(const T& t_) : t(t_) { }
}
    
```

```
};

// exemple d'utilisation
value<int> v(42);
```

Le problème ici est que `v` ne pourra stocker que des valeurs de type `int`. Or comme vu précédemment, un objet de type `boost::any` peut stocker un entier puis à la ligne suivante stocker une chaîne de caractères par exemple. Il nous faut donc pouvoir stocker des `value<T>` avec `T` variant d'un coup sur l'autre. Commençons par écrire une classe qui aura un `value<T>` et qui essayera de pouvoir charger une valeur de type différent sur demande.

```
class any
{
    value<T> v;
```

Voilà un problème. Quel `T` mettre ? Comment faire varier `T` d'un coup sur l'autre ? La programmation générique en elle même ne suffit plus. C'est là qu'intervient le principe de *Type Erasure*. Nous allons faire hériter tous les `value<T>` (donc la classe template) d'une même classe, et stocker un pointeur vers cette classe de base dans la classe `any`. Voilà donc ce que l'on obtient.

```
class value_base
{
public:
    virtual ~value_base() { }
};

template <class T>
class value : public value_base /* NOUVEAU */
{
    T t;

public:
    value(const T& t_) : t(t_) { }
};

class any
{
    value_base* v;

public:
    any() : v(0) { }

    template <class value_type>
    any(const value_type& v_) : v(new value<value_type>(v_)) { }

    ~any() { delete v; }
};

// exemple d'utilisation
{
    any a = 4;
    a = 'c';
}
```

Lors de la première ligne de l'exemple d'utilisation, on appelle le constructeur template. Pas de problème. Que se passe-t-il toutefois lors de la deuxième ligne ? En fait, un objet temporaire de type `any` va être construit pour encapsuler 'c' avec le constructeur template. Il est évident que ce "transfert" n'est pas très sécurisé du fait que l'opérateur d'assignation généré par défaut va "partager" le pointeur au lieu d'en retourner une copie. C'est pourquoi pour faciliter de telles opérations il va nous falloir une fonction dans `value<T>` pour cloner, c'est à dire construire une copie mais qui ne stockera pas un objet au même endroit de la mémoire, ainsi que le nécessaire pour l'assignation. Introduisons donc nos quelques modifications.

```
class value_base
```

```

{
    public:
    virtual ~value_base() { }
    virtual value_base* clone() const = 0; /* NOUVEAU */
};

template <class T>
class value : public value_base
{
    T t;

    public:
    value(const T& t_) : t(t_) { }
    value_base* clone() const /* NOUVEAU */
    {
        return new value(t);
    }
};

class any
{
    value_base* v;

    public:
    any() : v(0) { }

    template <class value_type>
    any(const value_type& v_) : v(new value<value_type>(v_)) { }

    any(any const & other) : v(other.v ? other.v->clone() : 0) {}

    any& operator=(const any& other)
    {
        if(&other != this)
        {
            any copy(other);
            swap(copy);
        }
        return *this;
    }

    void swap(any& other)
    {
        std::swap(v, other.v);
    }

    ~any() { delete v; }
};

```

A noter que nous utilisons l'idiome *Copy and Swap*, tel que présenté [ici](#).

Notre *any* réagit désormais correctement lors de copie depuis un autre *any*, ainsi que lors d'une assignation depuis un autre *any*. En effet, si vous avez testé le code présenté avant nos modifications, vous auriez vu que le problème mentionné plus haut menait à une erreur de segmentation. Désormais, tout se passe bien.

Ne reste plus maintenant qu'à implémenter le fameux *any_cast* qui permet d'essayer de récupérer la valeur contenue en explicitant le type de destination (puisque toute information de type a été "perdue", du moins publiquement). Il s'agit simplement ici de rendre cette fonction *any_cast* amie (plutôt que d'exposer notre *value_base* v* via une fonction publique) de sorte à ce qu'elle puisse tenter un *dynamic_cast* de *v* vers un *value<le type demandé>*. Nous allons donc ici n'écrire qu'une version de *any_cast*, celle qui prend un *any&* et renvoie le type demandé si la récupération réussit, une exception *bad_any_cast* le cas échéant.

```

class any; // pour permettre la déclaration suivante

template <class T>
T any_cast(any& a); // pour permettre les 'friend' dans les classes qui suivent

```

```

// modification de la classe template value
template <class T>
class value : public value_base
{
    friend T any_cast<>(any& a);
    // ...
};

// modification de la classe any
class any
{
    template <class T>
    friend T any_cast(any& a);
    // ...
};

// classe bad_any_cast
class bad_any_cast : public std::exception
{
public:
    const char* what() const throw()
    {
        return "Bad any_cast exception";
    }
};

// fonction template any_cast, version travaillant sur des références non constantes
template <class T>
T any_cast(any& a)
{
    value<T>* v = dynamic_cast<value<T>*>(a.v);
    if(v == 0)
    {
        throw bad_any_cast();
    }
    else
    {
        return v->t;
    }
}
    
```

Pour terminer, voici un code qui utilise tout ce que nous avons créé :

```

int main()
{
    any a = 42;
    any b = 'c';
    std::cout << "[1] a=" << any_cast<int>(a) << " b=" << any_cast<char>(b) << "" << std::endl;
    a.swap(b);
    std::cout << "[2] a=" << any_cast<char>(a) << " b=" << any_cast<int>(b) << std::endl;
    try
    {
        std::string s = any_cast<std::string>(b);
    }
    catch(const std::exception& e)
    {
        std::cout << "[3] " << e.what() << std::endl;
    }
    any c(a);
    std::cout << "[4] c=" << any_cast<char>(c) << "" << std::endl;
    return 0;
}
/*
alp@mestan:~/cpp$ g++ -o te3 type_erasure3.cpp
alp@mestan:~/cpp$ ./te3
[1] a=42 b='c'
[2] a='c' b=42
[3] Bad any_cast exception
    
```

```
[4] c='c'  
*/
```

Les 3 autres versions (références constantes, pointeurs non constants, pointeurs constants) sont laissées comme exercice pour le lecteur.

Nous sommes donc parvenus à une classe *any* faite maison accompagnée de la fonction template *any_cast* en appliquant simplement le principe de *Type Erasure*. Normalement, cela n'a pas été bien difficile, car une fois le principe connu, on sait qu'il nous suffit d'exposer l'interface minimale dans la classe de base de notre classe template afin de pouvoir parvenir à nos fins ensuite. Nous allons maintenant conclure quand aux limites de ce principe et à ce qui a été décrit ici.

VI - Limites et conclusion

Il n'y a qu'une vraie limite à ce type d'approche, qui est une fausse limite : on ne peut pas récupérer l'information que l'on a perdu sur le type précis de départ, comme *value<T>* pour notre classe *any*, que l'on se retrouve à traiter comme un *value_base*. Dans le cas des widgets, aucun moyen de savoir quels sont les types précis des widgets fils d'un certain widget à partir du moment où l'on les ajoute en tant que widget* dans la liste. Mais, bien évidemment, là est tout l'intérêt du polymorphisme de substitution ! Si l'on a créé cet héritage, c'est que que justement, quelque part, on voulait uniformiser un ensemble de valeurs et les traiter indifféremment, qu'il s'agisse de les stocker ensemble dans une collection comme pour les widgets, ou bien d'avoir une seule valeur mais qui peut prendre plusieurs valeurs de types concrets générés par une même classe template, pendant l'exécution.

Si vous voulez vous documenter un peu plus sur le principe de *Type Erasure* (sachez toutefois que cet article a couvert plus que l'essentiel sur le sujet), voici deux liens qui vous seront utiles.

- [Documentation du module Any de Boost 1.39](#)
- [On the Tension Between Object-Oriented and Generic Programming in C++](#), par Thomas Becker
- [An Efficient Variant Type](#), par Christopher Diggins

VII - Remerciements

Je tiens à remercier **Albert Pais**, **Florian Goujeon**, **dourouc05** particulièrement mais également le reste de l'équipe C++ pour leurs multiples relectures attentives.