

LE LANGAGE C++
MASTER I
QUELQUES IDIOMES

Jean-Baptiste.Yunes@u-paris.fr
UFR d'Informatique
Université de Paris

11/2021

IDIOMES

- traits
- policy
- SFINAE
- CRTP

IDIOMES

- un idiome est une forme d'expression relative à une communauté donnée
- en C++, de très nombreux idiomes sont disponibles qui permettent de réaliser certaines fonctionnalités
 - wikipedia en recense environ une centaine



LES TRAITIS

TRAITS

- cet idiome est parfois appelé the if-then-else of types
- un trait de type est une méta-information sur ce type
- les traits constituent une technique permettant d'obtenir, à la compilation, de l'information concernant des types génériques
 - il s'agit à la fois d'une stratégie très abstraite mais aux applications très concrètes

TRAITS

- avoir accès à des traits à la compilation permet de rédiger des programmes qui sont à la fois génériques et efficaces
- le compilateur ayant lui-même accès à de l'information sur les types, il peut procéder à des optimisations possibles

TRAITS

```
// abstraction d'un type
template <class T> struct Types {
    typedef T *pointeur;
    typedef T &reference;
};
...
class Vehicule {}
...
Types<int>::pointeur pi;
Vehicule v;
Types<Vehicule>::reference r = v;
```

TRAITS

```
// obtenir des informations utiles sur le type
template <class T> class Caracteristiques {
    static const bool is_numeric = false;
    static const int is_signed = false;
};
```


TRAITS

```
// spécialisation qui permet de définir les bonnes valeurs pour les traits...
template <> class Caracteristiques<int> {
    static const bool is_numeric = true;
    static const bool is_signed = true;
};

// spécialisation qui permet de définir les bonnes valeurs pour les traits...
template <> class Caracteristiques<unsigned short>
{
    static const bool is_numeric = true;
    static const bool is_signed = false;
};
```

TRAITS

```
#include <limits>

class TypeNonSigne {}

// la classe std::numeric_limits est une classe de traits
template <class T> T val_abs (const T &val)
{
    if (!std::numeric_limits<T>::is_signed)
        throw TypeNonSigne ();
    return (val < 0)? -val : val;
}
```

TRAITS

```
int main () {
    int i = -4;

    // Ok.
    std::cout << val_abs (i);
    unsigned int ui = 5;

    // Ici, lèvera TypeNonSigne
    std::cout << val_abs (ui);
    return 0;
}
```

TRAITS

```
// il est possible de définir les traits standardisés de son propre type :  
namespace std { // une extension du namespace std...  
    template <> numeric_limits<Voiture> {  
        enum { is_signed=false };  
    };  
}  
...  
Voiture v;  
val_abs(v); // une exception sera levée...
```

TRAITS

- mais encore ?
- supposons que l'on souhaite obtenir une optimisation consistant par exemple à ce que les variables primitives soit passées par valeur mais les objets par référence...
- l'idée est donc d'utiliser la spécialisation de template afin d'obtenir dans chaque cas le bon type à utiliser pour passer un argument

TRAITS

```
template <class T>
class optimized_arg {
private:
    template <class X,bool y=false> struct C {
        typedef const X &real_type; // le type pour les non primitifs
    };
    template <class X> struct C<X,true> {
        typedef X real_type; // le type pour les primitifs
    };
public:
    typedef typename
    C<T,is_primitive<T>::value>::real_type type;
};
```


TRAITS

```
template <class T> void f(typename optimized_arg<T>::type i) {
    i = 3; // tentative d'affectation...
}
class X {
public:
    X(int i=0) {}
};
int main() {
    X x;
    f<int>(4);
    f<X>(x); // erreur de compilation car const...
}
```

TRAITS

- dans l'exemple suivant on cherche à utiliser un trait pour :
 - déterminer si un type possède une certaine méthode
 - si oui, on l'utilise
 - si non, on utilise une alternative

TRAITS

```
// implémentation par défaut comme fonction...
template <bool internal>
struct selector {
    template <class T>
    static void aFunction(T &t) {
        cout << "fonction" << endl;
    }
};
```

TRAITS

```
// si le type supporte la bonne méthode...
template <>
struct selector<true> {
    template <class T>
    static void aFunction(T& t) {
        t.aMethod();
    }
};
```

TRAITS

```
// le trait par défaut...
template <class T>
struct supportsFunctionAsMethod {
    enum { value=false };
};

// la fonction qui fait l'optimisation...
template <class T>
void doTheTrick(T &t) {
    selector<supportsFunctionAsMethod<T>::value>::aFunction(t);
}
```

TRAITS

```
// un type ordinaire
struct standard {};
// un type qui implémente la méthode
struct withMethod {
    void aMethod() {
        cout << "method" << endl;
    }
};
// le trait qui correspond au type optimisé
template <>
struct supportsFunctionAsMethod<withMethod> {
    enum { value=true };
};
```


TRAITS

```
int main() {  
    standard s;  
    doTheTrick(s); // version externe  
    withMethod o;  
    doTheTrick(o); // version interne  
}
```

POLICY

POLICY

- les politiques (ou polices) sont des interfaces de classes qui à l'instar des traits fournissent au compilateur des informations sur les types. Le compilateur prendra donc des décisions concernant l'usage correct ou non des types considérés
- les politiques se préoccupent des aspects fonctionnels (les traits des aspects structurels)

POLICY

- les politiques permettent de paramétrer des types en ajoutant ou non des fonctionnalités particulières
- le compilateur réalisera les optimisations utiles
- comme pour les traits, ces constructions sont très efficaces (compile-time)

POLICY

- pour fixer les termes, il est utile de préciser :
 - une **politique** est une interface de classe ou une interface de classe template
 - une **classe de politique** est une implémentation (une réalisation) d'une politique
 - les classes qui utilisent une classe de politique sont appelées **classes hôtes**

POLICY

```
template <class Politique>
class Hote {
public:
    void f() { Politique::f(); }
    void g() { Politique::g(); }
};
```


POLICY

```
class Politique1 {
public:
    void f() { ... }
};
class Politique2 {
public:
    void f() { ... }
    void g() { ... }
};
```

POLICY

- une particularité des *templates* (que nous étudierons ensuite) est de ne pas provoquer d'erreur lorsqu'un code *template* qui est syntaxiquement correct mais logiquement incorrect n'est pas appelé, ainsi :
- la classe **Hote<Politique1>** est utilisable à condition de ne jamais faire appel à la méthode **g ()**
- on peut utiliser librement **f ()** et **g ()** de la classe **Hote<Politique2>**

POLICY

- on notera qu'une telle construction ressemble fortement au pattern *strategy*. C'est effectivement le cas, et celui-ci est aussi appelé *policy pattern*...

POLICY

```
template <class Politique>
class Hote : public Politique {
public:
    typename Politique::type g() {
        return typename Politique::type(5);
    }
};
class P1 {
public:
    typedef int type;
    static type f() { return type(3); }
};
```

POLICY

```
int main() {  
    Hote<P1> h;  
    cout << h.f() << endl;  
    cout << h.g() << endl;  
}
```

POLICY

- dans ce cas on utilise l'héritage pour supporter la politique
 - la structure de la classe hôte en est modifiée, donc son comportement (ce qui est le but principal des politiques)

POLICY

- en un certain sens on a renversé la situation par rapport à l'héritage :
 - **héritage** : la classe au sommet est abstraite et on l'implémente en descendant la hiérarchie
 - ouverture par le bas
 - **policy** par héritage : la classe hôte (en bas) est abstraite, elle est implémentée en héritant des classes politiques transmises
 - attention car dans ce cas, l'héritage ne correspond pas à la relation de généralisation/spécialisation, *i.e.* ce n'est pas est-un / is-a
 - ouverture par le haut

POLICY

```
template <class T,  
        template <class I> class Politique>  
class Hote : public Politique<T> {  
public:  
    typename Politique<T>::type g() {  
        return typename Politique<T>::type(5)/2;  
    }  
};
```

POLICY

```
template <class T>
class P1 {
public:
    typedef T type;
    static type f() { return type(3); }
};
```

POLICY

```
int main() {  
    Hote<float,P1> h;  
    cout << h.f() << endl;  
    cout << h.g() << endl;  
    Hote<int,P1> h2;  
    cout << h2.f() << endl;  
    cout << h2.g() << endl;  
}
```

POLICY

- ici la classe de politique est elle-même une classe *template*...
 - on a donc, une classe hôte *template* qui hérite d'une classe de politique *template*

POLICY

- dans l'exemple suivant, on va utiliser des politiques pour réaliser un « Hello World » paramétré :
 - la langue sera une politique (fonction qui renvoie le bon message)
 - la fonction de sortie sera une politique

POLICY

```
template < typename output_policy,  
           typename language_policy >  
class HelloWorld : public output_policy,  
                  public language_policy {  
public:  
    void Run() {  
        output_policy::Print(language_policy::Message());  
    }  
};
```

POLICY

```
// la politique de sortie sur écran...
class HelloWorld_OutputPolicy_WriteToCout {
protected:
    template< typename message_type >
    void Print( message_type message ) {
        cout << message << endl;
    }
};
```


POLICY

```
// la politique de langue (anglais)
class HelloWorld_LanguagePolicy_English {
protected:
    string Message() {
        return "Hello, World!";
    }
};

// la politique de langue (français)
class HelloWorld_LanguagePolicy_French {
protected:
    string Message() {
        return "Bonjour tout le monde!";
    }
};
```

POLICY

```
// en Anglais sur l'écran
int main() {
    typedef
    HelloWorld<HelloWorld_OutputPolicy_WriteToCout,
               HelloWorld_LanguagePolicy_English>
    my_hello_world_type;

    my_hello_world_type hello_world;
    hello_world.Run();

    return 0;
}
```

POLICY


```
// en Français à l'écran
int main() {
    typedef
    HelloWorld<HelloWorld_OutputPolicy_WriteToCout,
               HelloWorld_LanguagePolicy_French >
    my_other_hello_world_type;

    my_other_hello_world_type hello_world2;
    hello_world2.Run()
    return 0;
}
```

POLICY

- dans ce fameux programme revisité, on emploie deux politiques
 - ainsi plus le nombre de paramètres de type est grand plus la combinatoire est élevée, la richesse d'expression se révèle...

POLICY

 remarque : attention à faire en sorte que les politiques soient orthogonales, *i.e.* que le chevauchement conceptuel/fonctionnel des politiques soit nul (ou quasi-nul)

- dans le cas contraire, l'instanciation de la classe hôte devient trop difficile
 - on notera que dans le cas normal, c'est déjà bien compliqué...
 - la documentation est **ABSOLUMENT** nécessaire

SFINAE

SFINAE

- SFINAE (Substitution Failure Is Not An Error) signifie que lorsque le compilateur substitue les types dans la déclaration d'une fonction *template*, et que cette substitution échoue, ceci ne constitue pas une erreur :
- le compilateur continue en recherchant d'autres *templates*, jusqu'à ce qu'une substitution soit possible (si finalement il n'y en a pas alors cela devient une erreur)

SFINAE

- autrement dit, si la substitution des types dans une fonction *template* échoue, ce *template* est discrètement éliminé du jeu, sans signalement d'erreur

SFINAE

```
template < typename T >
void f(typename T::type) { // première définition
}

template < typename T >
void f(T) {}                // seconde définition

struct Test {
    typedef int type;
};

int main(){
    f<Test>(10); // 1
    f<int>(10);  // 2
}
```

SFINAE

```
template<bool B, typename T = void>  
struct my_if {};
```

```
template<typename T>  
struct my_if<true,T> {  
    typedef T type;  
};
```

my_if<V,T>::type vaut
T quand V est true, mais
n'existe pas quand T vaut
false

SFINAE

- exemple d'une fonction qui n'est définie que sur les types entiers :

```
template<typename T>
typename
    my_if<std::is_integral<T>::value, T>::type
    uneFonction(T x);
```

SFINAE

- `my_if` pouvant être utilisé avec tout type de *template*, il permet d'effectuer des décisions pour autoriser ou non certains *templates*
- il faut qu'un seul *template* surchargé soit actif pour un jeu donné d'arguments fournis au *template*

SFINAE

```
// vecteur dans l'espace à deux dimensions
class My_vector {
    int _x, _y;
public:
    My_vector(int x, int y) : _x(x), _y(y) {}
    int x() const { return _x; }
    int y() const { return _y; }
};
```

SFINAE

```
// point dans l'espace à deux dimensions
class My_point {
    int _x, _y;
public:
    My_point(int x, int y) : _x(x), _y(y) {}
    int x() const { return _x; }
    int y() const { return _y; }
};
```


SFINAE

```
// définition d'un prédicat...  
// classe de traits qui détermine si T est un vecteur 2D  
template < typename T >  
struct IsVector {  
    enum { value = false }; // par défaut, T n'est pas vecteur  
};  
  
// spécialisation du template précédent pour My_vector  
template <>  
struct IsVector <My_vector> {  
    enum { value = true }; // mais My_vector est un vecteur  
};
```

SFINAE

```
// définition conditionnée de l'opérateur d'addition
template < typename V >
typename my_if<IsVector<V>::value, V>::type
    operator+(V const &v, V const &w) {
        return V (v.x() + w.x(), v.y() + w.y());
    }
int main() {
    My_vector v(1,2), w(3,4);
    My_vector z = v + w;    // OK
    My_point p(1,2), q(3,4);
    // erreur : pas de correspondance avec l'opérateur My_point& +
    My_point& r = p + q;
}
```

C RTP

CRTP

CRTP (Curiously Recurring Template Pattern) désigne un motif d'héritage dans lequel une classe *template* de base reçoit comme valeur pour son paramètre *template* une de ses classes dérivées :

- cela revient à dire qu'à l'instanciation la classe de base peut connaître ses sous-classes

C RTP

- cette technique permet d'obtenir le même effet que des méthodes virtuelles, sans le coût associé (*static polymorphism*)
- elle peut être utilisée pour obtenir une notion de conformité vis-à-vis d'une interface avec génération de code et un typage correct

CRTP

```
template <class T>
class BASE {
public:
    void interface() {
        static_cast<T *>(this)->impl();
    }
};
class DÉRIVÉE : public BASE<DÉRIVÉE> {
public:
    void impl() {
        cout << "DÉRIVÉE::impl()" << endl;
    }
};
```

étrange non ?

C RTP

```
class DÉRIVÉE2 : public BASE<DÉRIVÉE2> {
public:
    void impl() {
        cout << "DÉRIVÉE2::impl()" << endl;
    }
};

int main() {
    DÉRIVÉE d;
    d.interface(); // appel « dynamique »
    DÉRIVÉE2 d2;
    d2.interface(); // appel « dynamique »
    return 0;
}
```

C RTP

- un autre exemple

CRTP

```
// Afficher ne sera définie que pour les types dérivés...  
  
template <class E> class Affichable {  
public:  
    friend void Afficher (const E &e) {  
        cout << e << endl;  
    }  
};
```

C RTP

```
// les entiers sont affichables...
class Entier : public Affichable<Entier> {
    int m_Val;
public:
    Entier (const int Val) : m_Val(Val) {
    }
    int getVal () const {
        return m_Val;
    }
};
```

C RTP

```
std::ostream& operator<<(std::ostream &os,  
                          const Entier &e)  
{  
    return os << e.getVal ();  
}  
  
int main () {  
    Entier e(3);  
    Afficher(e);  
}
```

CRTP

- la classe parent (générique) se voit injecter à l'instanciation (au sens de la génération par le compilateur de la classe à laquelle s'applique le *template*) le type de son enfant.
- cette technique est possible parce que le nom de l'enfant apparaît avant le nom du parent et existe donc au moment où le parent est généré par le compilateur

CRTP

- ainsi la sélection de la méthode à appeler est réalisée statiquement (à la compilation), il n'y a donc pas de surcoût lié au dynamisme (*static polymorphism*)

CRTP

```
namespace relation {  
// equivalence<T> définit une relation d'équivalence pour T si T::operator<(const T&) const existe  
template <class T> class equivalence {  
public:  
    friend bool operator>(const T &a,const T &b) {  
        return b < a;  
    }  
    friend bool operator<=(const T &a,const T &b) {  
        return !(b < a);  
    }  
    friend bool operator>=(const T &a,const T &b) {  
        return !(a < b);  
    }  
};  
}
```

C RTP

```
// un nombre est conforme à la relation d'équivalence
class nombre : public
    relation::equivalence<nombre> {
    int m_Valeur;
public:
    nombre (const int n = 0) : m_Valeur (n) { }
    int getValeur () const {
        return m_Valeur;
    }
    bool operator< (const nombre &n) const {
        return getValeur () < n.getValeur ();
    }
};
```

CRTP

```
int main () {
    nombre n0(4), n1(4);
    if (n0 < n1) cout << n0.GetValeur() << "< "
                  << n1.GetValeur() << endl;
    if (n0 <= n1) cout << n0.GetValeur() << "<="
                      << n1.GetValeur() << endl;
    if (n0 > n1) cout << n0.GetValeur() << "> "
                  << n1.GetValeur() << endl;
    if (n0 >= n1) cout << n0.GetValeur() << ">="
                    << n1.GetValeur() << endl;
}
```


CRTP

// La classe de base déclare une méthode de clonage
on devrait définir dans toutes les sous-classes une
méthode de clonage de type

```
Animal {  
public:  
    virtual Animal *clone() const = 0;  
};
```

CRTP

```
// on créé une classe intermédiaire qui implémente clone pour toutes  
// les sous-classes...
```

```
template <typename T>  
class AnimalClonable: public Animal {  
public:  
    Animal *clone() const {  
        return new T(dynamic_cast<T const &>(*this));  
    }  
};
```

C RTP

```
// désormais les sous-classes « héritent » d'une méthode de clonage  
générique qui renvoie le bon clone pour chaque sous-classe  
class Chien: public AnimalClonable<Chien> {  
};  
  
class Chat: public AnimalClonable<Chat> {  
};  
  
class Poisson: public AnimalClonable<Poisson> {  
};
```