



LE LANGAGE C++ MASTER I LA MÉTA-PROGRAMMATION

Jean-Baptiste.Yunes@univ-paris-diderot.fr

U.F.R. d'Informatique

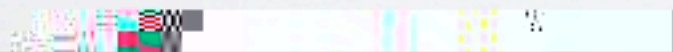
Université Paris Diderot - Paris 7

nov. 2012

- Bon programmer ok,
- mais méta-programmer ?
- méta fait toujours référence à un niveau supérieur...

- données, méta-données
 - programme, méta-programme ?
- méta-données décrivent des données
 - un méta-programme décrit un programme ?
- méta-données sont interprétées (à travers une exécution) pour manipuler des données
 - méta-programme est interprété/exécuté afin de produire un programme ?

- on oublie facilement que les compilateurs sont dotés de capacités de calcul
- ils implémentent un(des) algorithme(s) permettant de traduire un langage dans un autre
- mais au passage il font de l'**evaluation partielle**
- ils exécutent une partie du code à compiler
- afin d'**optimiser**



- Si l'on écrit en C ou en C++ :

```
a = 3+8+100;
```

- le compilateur génère un code correspondant à

```
a = 111;
```

```
int main() {  
    int a;  
    a = 3+8+100;  
}
```

C



gcc -S

```
LCFI1:  
    movl    $111, -4(%rbp)  
    leave  
    ret
```

i386

- Mon compilateur sait-il faire quelque chose d'autre ?
 - mise à part la compilation proprement dite...
 - ...des optimisations qui sont toutes des évaluations partielles
- considérons les templates...
 - l'instanciation d'un template, qu'est ce ?
 - similaire à un appel de fonction, non ?

- Comment s'instancie un template ?
 - on détermine la valeur des arguments lors de l'instanciation, ces valeurs ne peuvent être que des constantes (**type** ou valeur numérique)
 - on définit un code correspondant à celui décrit par le template en remplaçant les variables du template par les valeurs passés
- L'*instanciation* (exécution) peut produire un **type** ou une **fonction**

- 
- 
- c'est donc une forme de programmation
 - qui permet de travailler sur
 - des types
 - des fonctions


```
template <int V> class G {  
};
```

```
G<30> unObjet;
```

valeur produite à la compilation : G appliqué à 30

```
template <class T> class F {  
};
```

```
F<int> unAutreObjet;
```

valeur produite à la compilation : F appliqué à int

```
template <class T> T *create() {  
    return new T;  
};
```

```
int *pi = create<int>();
```

valeur produite à la compilation : create appliqué à int

Appel simple de fonction

```
template <int V> class G {  
public:  
    static const int valeur = V;  
};  
int x = G<30>::valeur;
```

Appel simple de fonction

valeur de constante produite à la compilation

valeur de type produite à la compilation
résultat de G appliqué à 30

- D'évidence, cette programmation est un peu particulière
 - vraiment ???
- sa syntaxe est un peu étrange...
 - ...comme tous les langages, non ?

```
template <int V> class F {  
public:  
    static const int valeur = V*V;  
};
```

```
template <int V> class G {  
public:  
    static const int valeur = V+F<V>::valeur;  
};
```

```
int X = G<30>::valeur;
```

Une fonction appelle
une autre fonction


```
template <int V> class Bizarre {  
public:  
    static const  
        int valeur = V+Bizarre<V-1>::valeur;  
};
```

Appel récursif

Comment un tel appel peut-il terminer ?

rappel : une spécialisation de template
l'emporte toujours à l'instanciation

```
template <int V> class Bizarre {  
public:  
    static const  
        int valeur = V+Bizarre<V-1>::valeur;  
};
```

Appel récursif

```
template <> class Bizarre<0> {  
public:  
    static const int valeur = 0;  
};
```

Terminaison

```
int main() {  
    int a = Bizarre<10>::valeur;  
}
```

Exercice : combien cela vaut-il ?

```
template <int C,class T,class E>  
class IF {};
```

```
template <class T,class E>  
class IF<true,T,E> {  
public:  
    typedef T type;  
};
```

```
template <class T,class E>  
class IF<false,T,E> {  
public:  
    typedef E type;  
};
```

Alternative

```
template <int C, class T, class E>
class IF {};
```

```
template <class T, class E>
class IF<true, T, E> {
public:
    typedef T type;
};
```

```
template <class T, class E>
class IF<false, T, E> {
public:
    typedef E type;
};
```

Ce type est déterminé par un calcul (effectué à la compilation)

```
int main() {
    const bool b = false;
    IF<b, int, float>::type v;
    v = 3.5;
    cout << v << endl;
    return 0;
}
```


Alternative

```
template <int X> class SOMME {  
public:  
    static const int valeur = X+SOMME<X-1>::valeur;  
};  
template <> class SOMME<0> {  
public:  
    static const int valeur = 0;  
};
```

une fonction

```
template <int C, class IF> {  
class IF {};
```

```
template <class T, class E>  
class IF<true, T, E> {
```

```
public
```

```
    type  
};
```

```
templa
```

```
class
```

```
public
```

```
    type
```

```
};
```

```
int main() {  
    IF<SOMME<10>::valeur%2, int, float>::type v;  
    v = 3.5;  
    cout << v << endl;  
    return 0;  
}
```

- mais encore ?
 - et les structures de données ?
 - on manipule des types...
 - peut-on manipuler des structures dont les éléments seraient des types ?
 - ...oui

- une liste de types
- facile

Structure de données : la liste

définition du type
« liste de types »

```
template <class ELEMENT, class SUITE>
class ListeDeTypes {
public:
    typedef ELEMENT type;
    typedef SUITE suite;
};
```

une variable de type
« liste de types »

une liste de types
(int, float, A)

on définit un type

```
class A {};
```

```
typedef ListeDeTypes<int,
    ListeDeTypes<float,
        ListeDeTypes<A, void> > > MaListe;
```

- on peut manipuler cette liste
- comme d'habitude...

Structure de données : la liste

```
class A {};  
  
typedef ListeDeTypes<int,  
    ListeDeTypes<float,  
        ListeDeTypes<A,void> > > MaListe;
```

```
MaListe::suite::type uneVariable;  
MaListe::suite::suite::type uneAutreVariable;
```

de type float

de type A

- ok, mais comment obtenir le 3^{ième} ?
- le $n^{\text{ième}}$?
- il nous faut une itération...

```
template <class L,int N,int M=1>
class Iteration {
public:
    typedef Iteration<L::suite,N,M+1>::type type;
};

template <class L,int N>
class Iteration<L,N,N> {
public:
    typedef L::type type;
};
```

Bonne idée, mais le compilateur n'est pas content... En effet, s'agit-il de constantes de classe (variable) ou de types ?

```
template <class L,int N,int M=1>
class Iteration {
public:
    typedef typename Iteration<typename L::tail,N,M+1>::type type;
};

template <class L,int N>
class Iteration<L,N,N> {
public:
    typedef typename L::type type;
};
```

typename : mot-clé du C++ permettant d'indiquer que ce qui suit correspond bien à un type. Ce mot-clé ne peut être employé que dans le contexte d'un template...

```
template <class L,int N,int M=1>
class Iteration {
public:
    typedef typename Iteration<typename L::tail,N,M+1>::type type;
};
```

```
template <class L,int N>
class Iteration<L,N,N> {
public:
    typedef typename L::type type;
};
```

Attention à bien
mettre des espaces
pour ne pas confondre
avec >>

```
typedef ListeDeTypes<int,
    ListeDeTypes<float,
        ListeDeTypes<A,void> > > MaListe;
```

```
Iteration<MaListe,3>::type uneVariable;
Iteration<MaListe,1>::type uneAutreVariable;
```

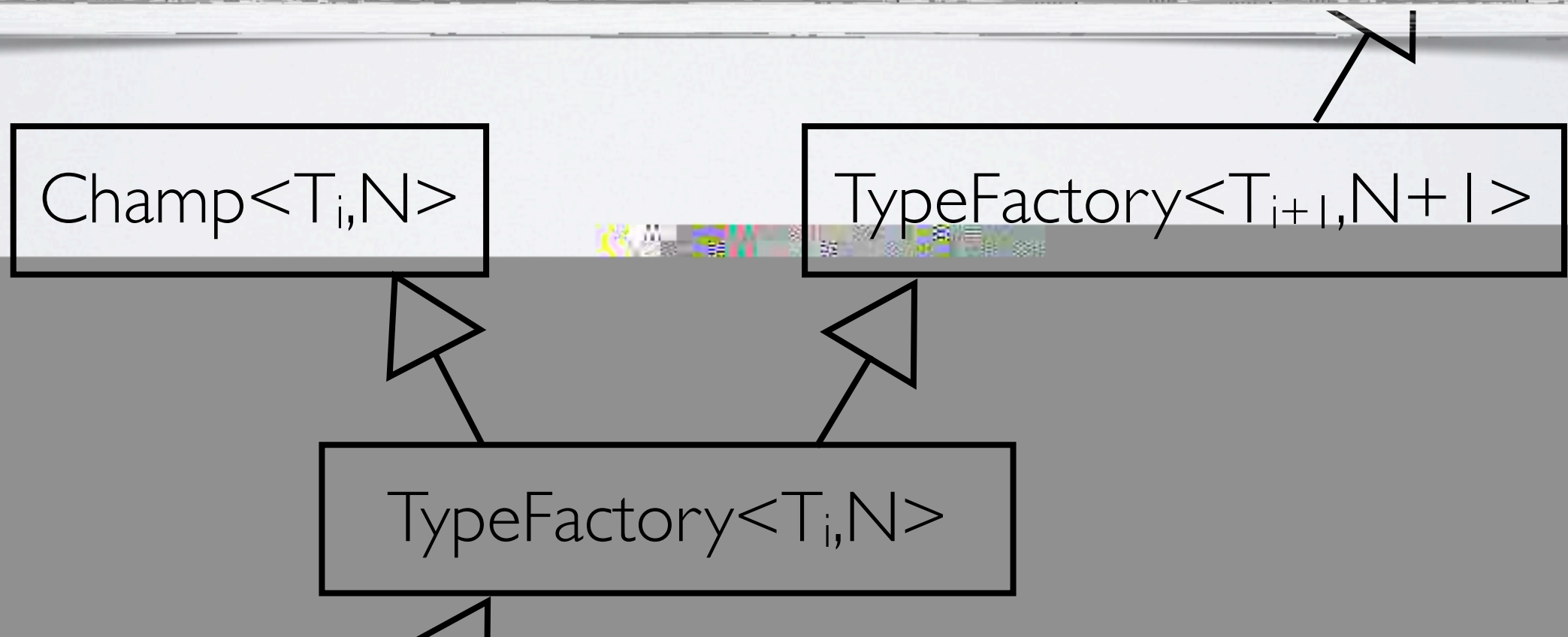
- fabrication *automatique* d'une structure contenant des champs dont les types appartiennent à une liste donnée
- il suffit de *mémoriser* les différents types lors de l'itération
- la *mémorisation* pouvant consister à fabriquer une arbre d'héritage
- chaque niveau contenant un champ de type correspondant dans la liste...

- Tout d'abord, une structure permettant d'encapsuler un champ d'un type donné
 - tatoué par un entier afin de lever les ambiguïtés

```
template <class T,int N> class Champ {  
    private:  
        T value;  
    public:  
        T      &getValue()          { return value; }  
        void setValue(const T&v) { value = v; }  
};
```

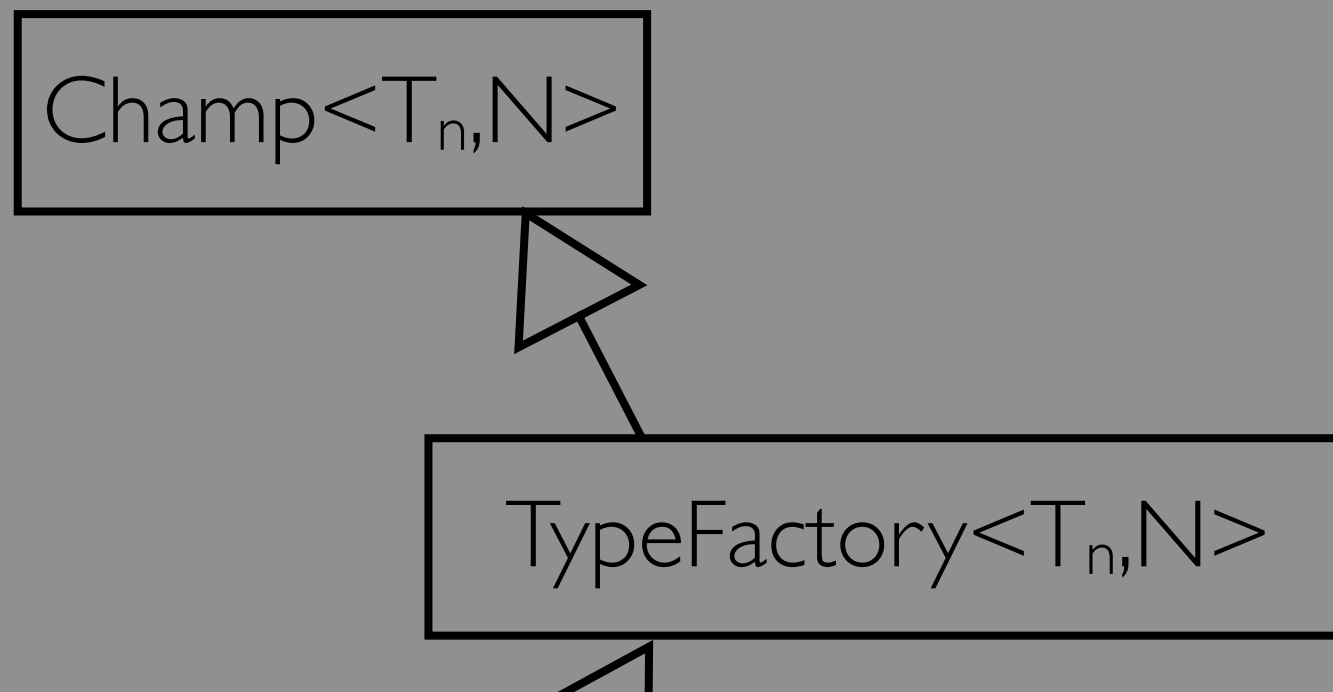
- le cas général de l'itération
 - instantiation d'un champ
 - appel récursif sur le type suivant de la liste...

```
template <class L,int N=1>
class TypeFactory :
    public TypeFactory<typename L::suite,N+1>,
    public Champ<typename L::type,N> {
};
```



- le cas terminal de l'itération
 - détection du dernier type de la liste par spécialisation
 - instantiation d'un champ

```
template <class T,int N>  
class TypeFactory<TypeList<T,void>,N > :  
    public Champ<T,N> {  
};
```



- reconstruction du type du $n^{\text{ième}}$ champ
- par itération sur la liste de type

```
template <class L,int N,int M=1>
class TypeDuChamp {
public:
    typedef
        Champ<typename Iteration<L,N,M>::type,N>
        type;
};
```


- on peut donc utiliser librement le *up-casting* pour obtenir le bon champ...

```
typedef ListeDeTypes<int,<ListeDeTypes<float,<ListeDeTypes<Z,void>  
> > lt;
```

```
TypeFactory<lt> uneStructure;
```

```
TypeDuChamp<lt,2>::type unChamp =  
    static_cast<TypeDuChamp<lt,2>::type &>(s);  
cout << unChamp.getValue() << endl;
```


- bien entendu on peut continuer...
- ...par exemple pour fabriquer des hiérarchies parallèles de classes
 - c'est important pour le *pattern* visitor!

- La bibliothèque BOOST