

# LE LANGAGE C++

## MASTER 1

### QUELQUES TECHNIQUES AVANCÉES

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

UFR d'Informatique

Université Paris Diderot - Paris 7

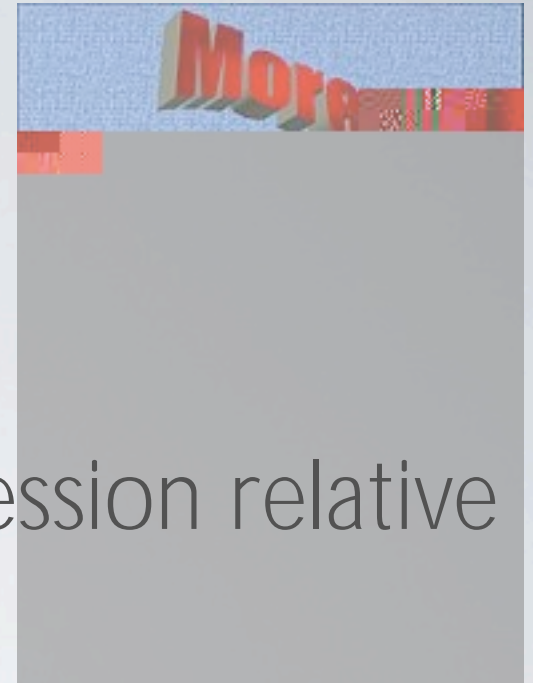
Déc. 2012

# 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 recense environ une centaine



# LES TRAITES

# 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

# TRAITS

```
// obtenir des informations utiles sur le type
```



# TRAITS

spécialisation qui permet de définir les bonnes valeurs pour les traits...

spécialisation qui permet de définir les bonnes valeurs pour les traits...

# TRAITS

la classe

est une classe de traits

# TRAITS

# TRAITS

il est possible de définir les traits standardisés de son propre type :  
une extension du namespace std...

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

le type pour les non primitifs

le type pour les primitifs

# TRAITS

tentative d'affectation...

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...

# TRAITS

si le type supporte la bonne méthode...

# TRAITS

le trait par défaut...

la fonction qui fait l'optimisation...

# TRAITS

un type ordinaire

un type qui implémente la méthode

le trait qui correspond au type optimisé

# TRAITS

version externe

version interne

# POLICY

# POLICY

- les politiques (ou policies) 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 structures)

# POLICY

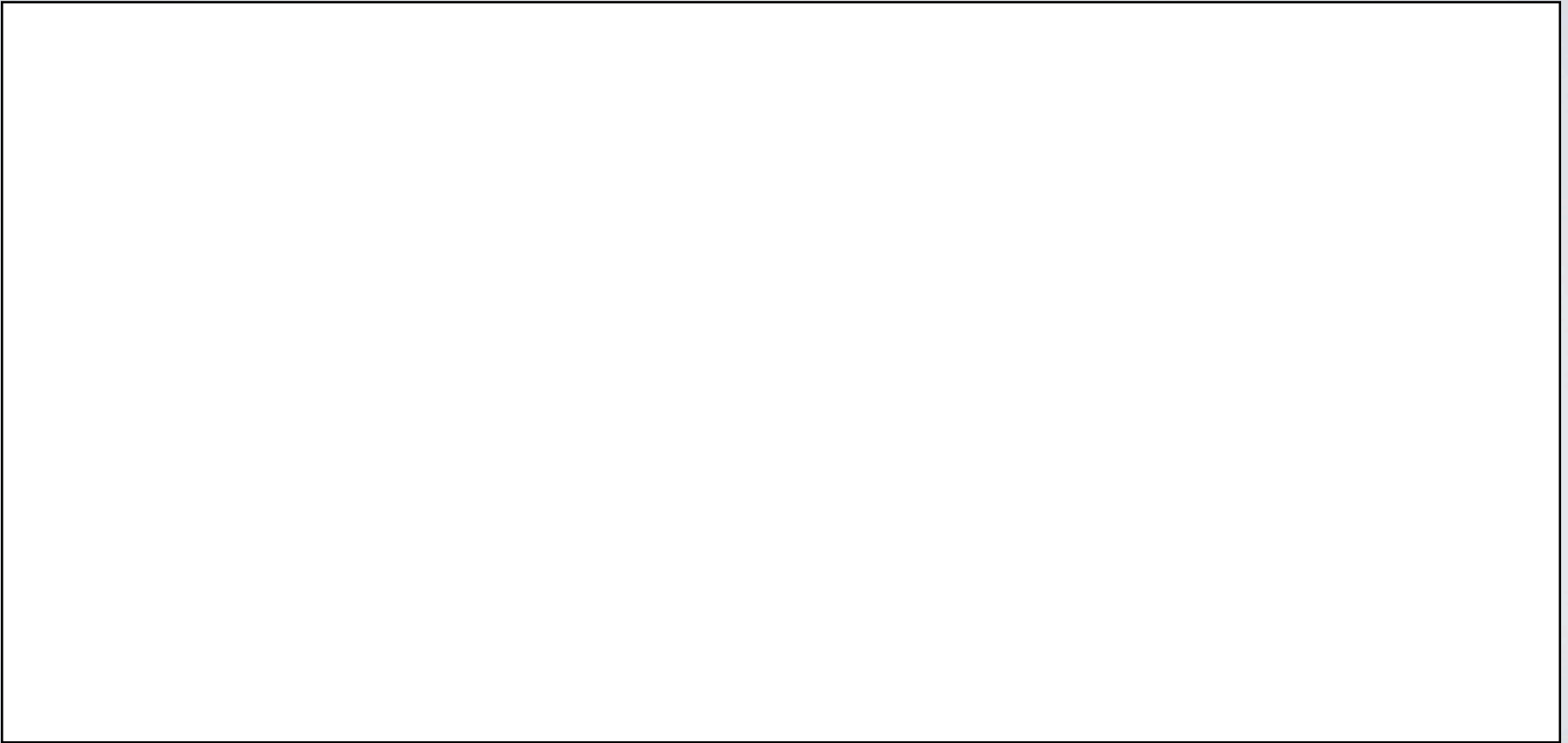
- les politiques permettent de paramétrer des types en ajoutant ou non des fonctionnalités particulières
- le compilateur réalisera les optimisations utiles
- comme pour les traits, ces constructions sont très efficaces (compile-time)



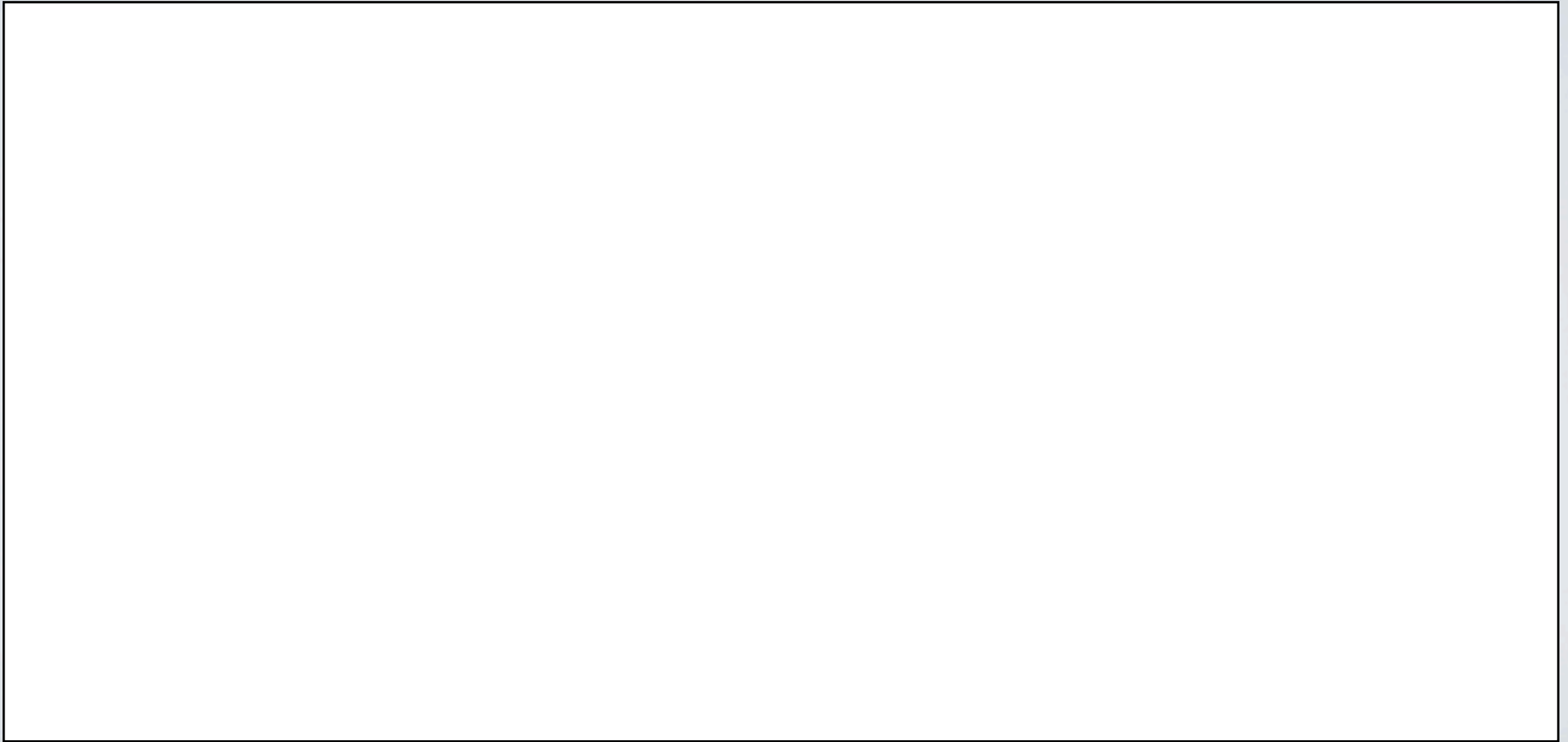
# POLICY

- 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



# POLICY



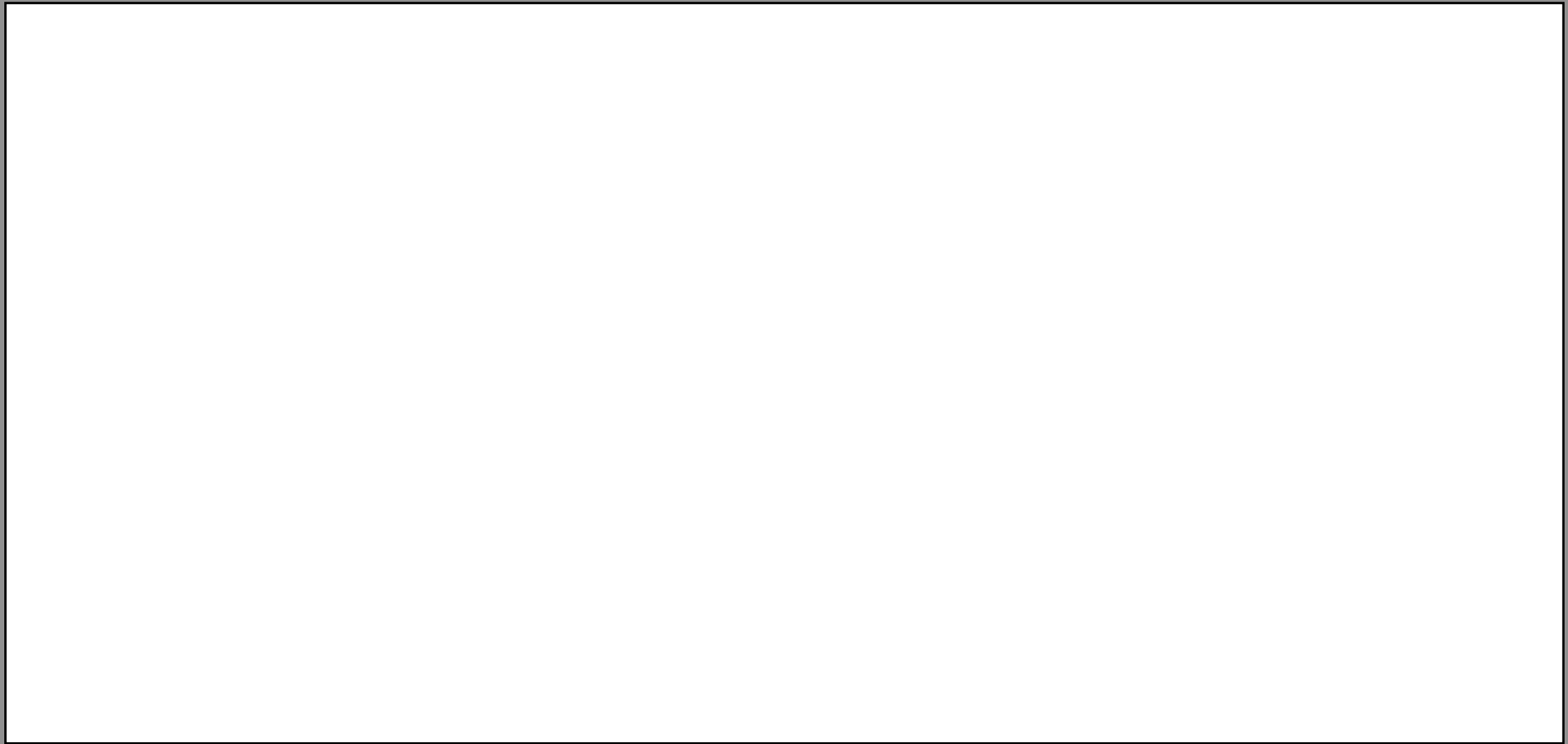
# POLICY

- 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 `std::enable_if` est utilisable à condition de ne jamais faire appel à la méthode `enable_if::type`
- on peut utiliser librement `std::enable_if` et `std::enable_if_t` de la classe `std::enable_if`

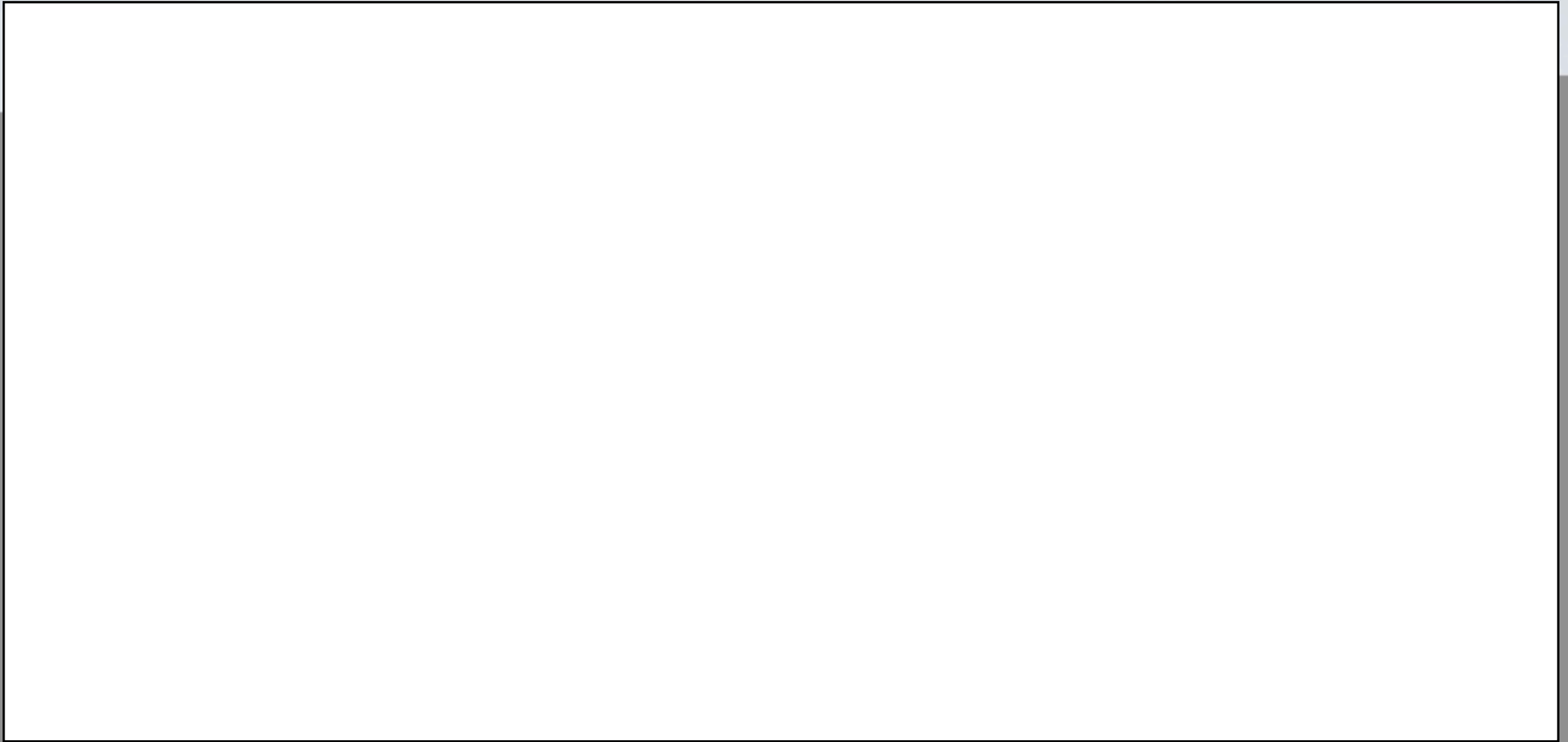
# POLICY

- 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



# POLICY



# POLICY

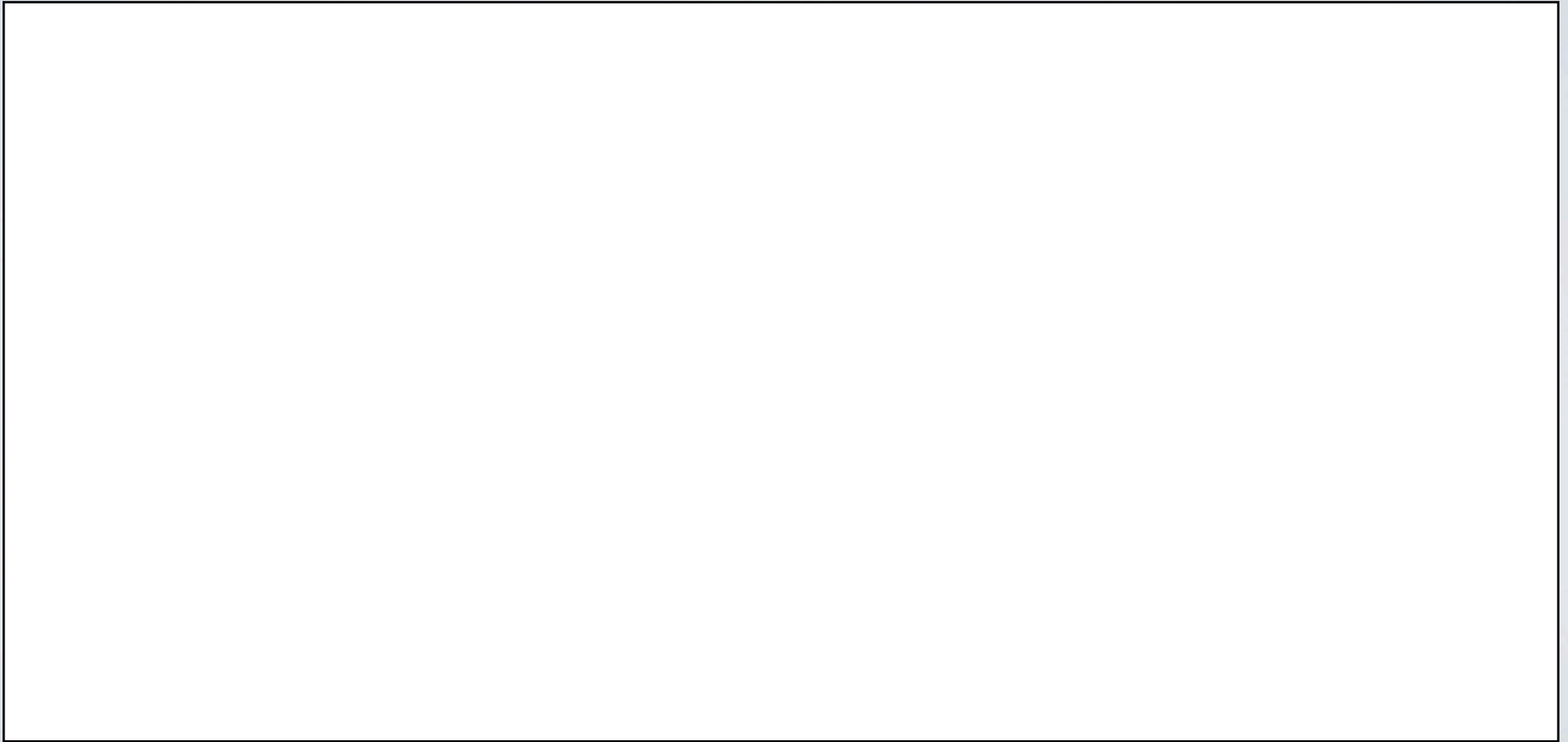
- dans ce cas on utilise l'héritage pour supporter la politique
  - la structure de la classe hôte en est modifiée, donc son comportement (ce qui est le but principal des politiques)



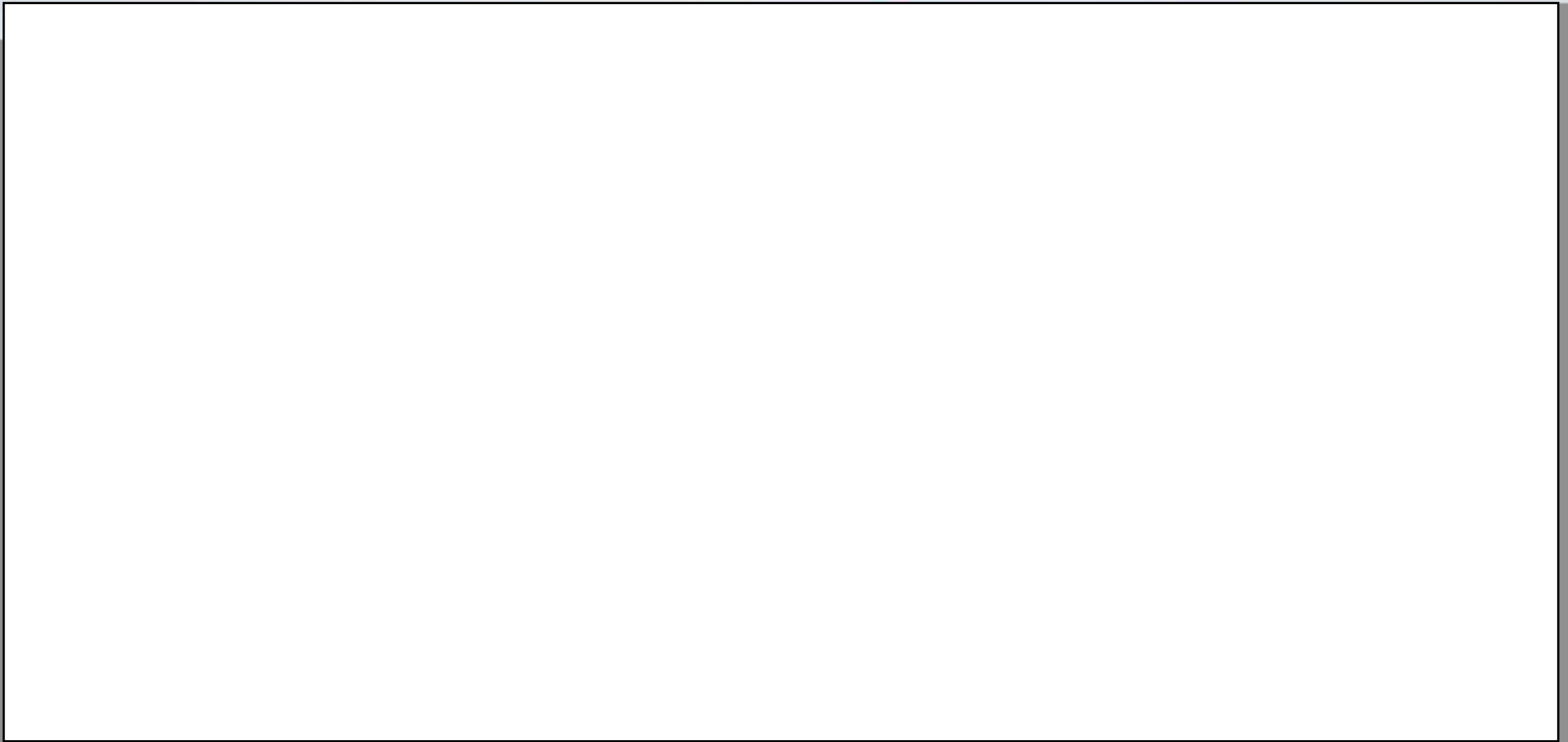
# POLICY

- 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
  - **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

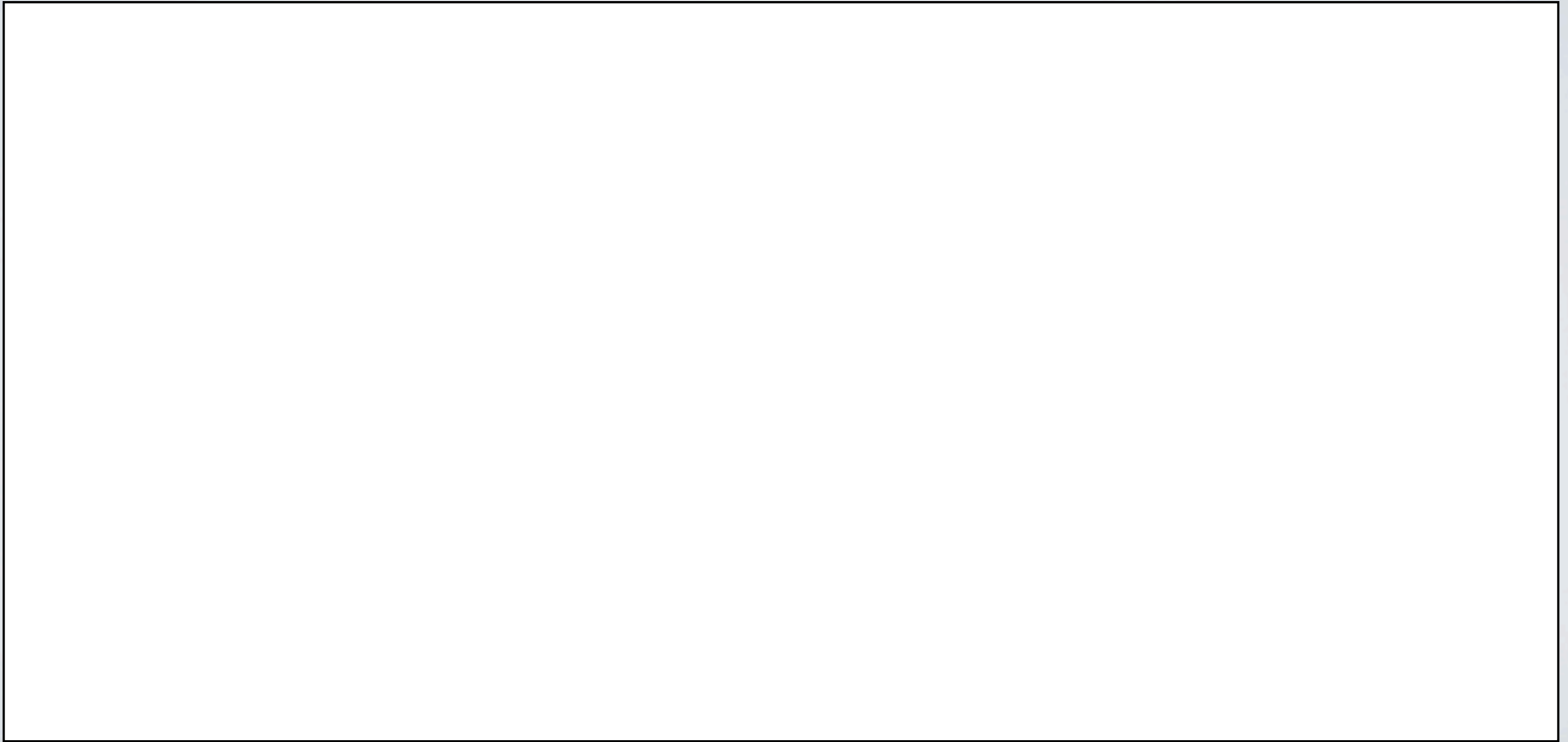
# POLICY



# POLICY



# POLICY

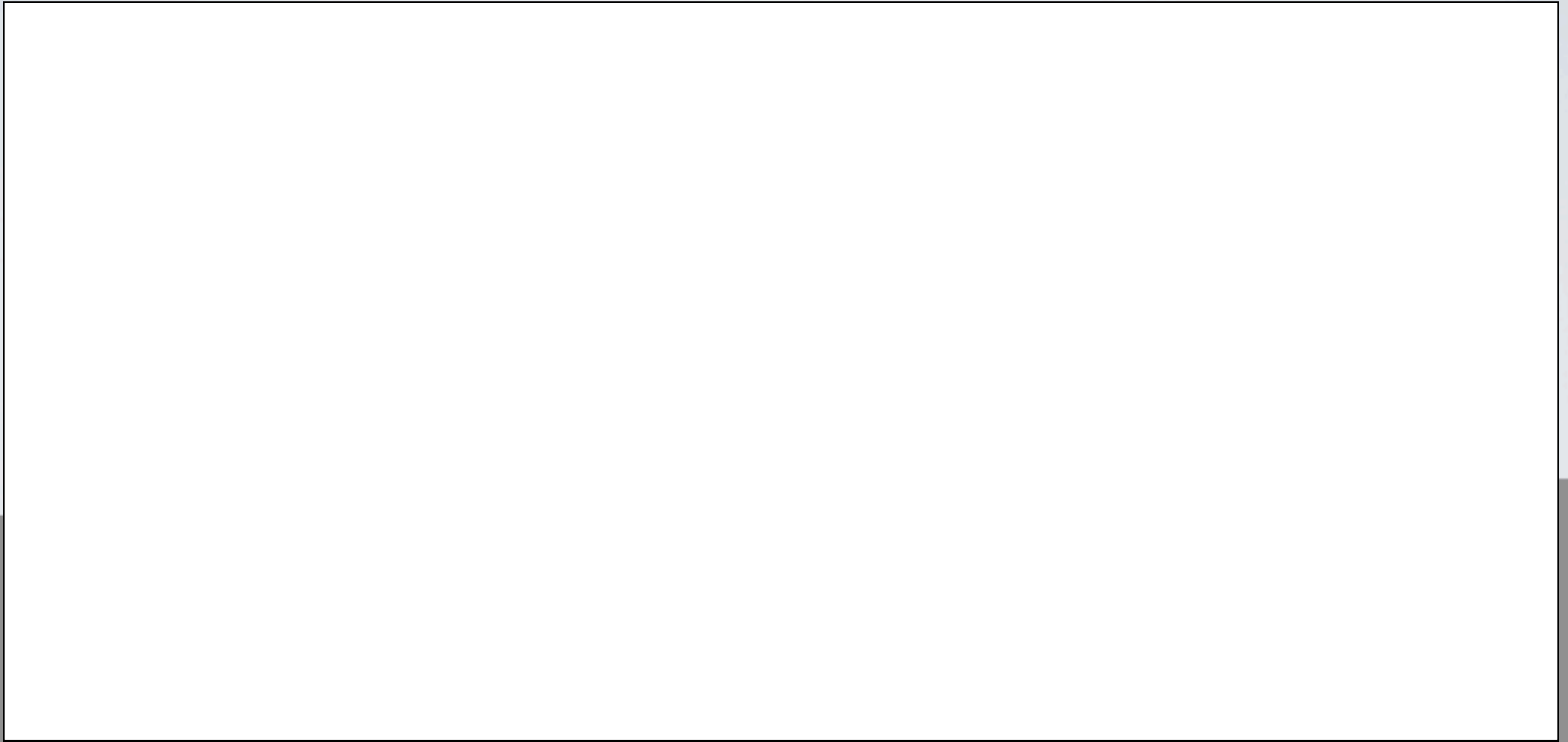




# POLICY

- 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



# POLICY

la politique de sortie sur écran...



# POLICY

la politique de langue (anglais)

la politique de langue (français)

# POLICY

en Anglais sur l'écran

# POLICY

en Français à l'écran

# 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

première définition

seconde définition

# SFINAE

vaut  
quand est , mais  
n'existe pas quand vaut

# SFINAE

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

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

# SFINAE

- 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

# SFINAE

point dans l'espace à deux dimensions

# SFINAE

définition d'un prédicat...

classe de traits qui détermine si `typename T::value_type` est un vecteur 2D

par défaut, `typename T::value_type` n'est pas vecteur

spécialisation du template précédent pour

mais `typename T::value_type` est un vecteur

# SFINAE

définition conditionnée de l'opérateur d'addition

OK

erreur : pas de correspondance avec l'opérateur





CRTP

# 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

# CRTP

- 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

# C RTP

**BASE**

**DÉRIVÉE**

**BASE<DÉRIVÉE>**

étrange non ?

# C RTP

appel « dynamique »

appel « dynamique »

# C RTP

- un autre exemple

# C RTP

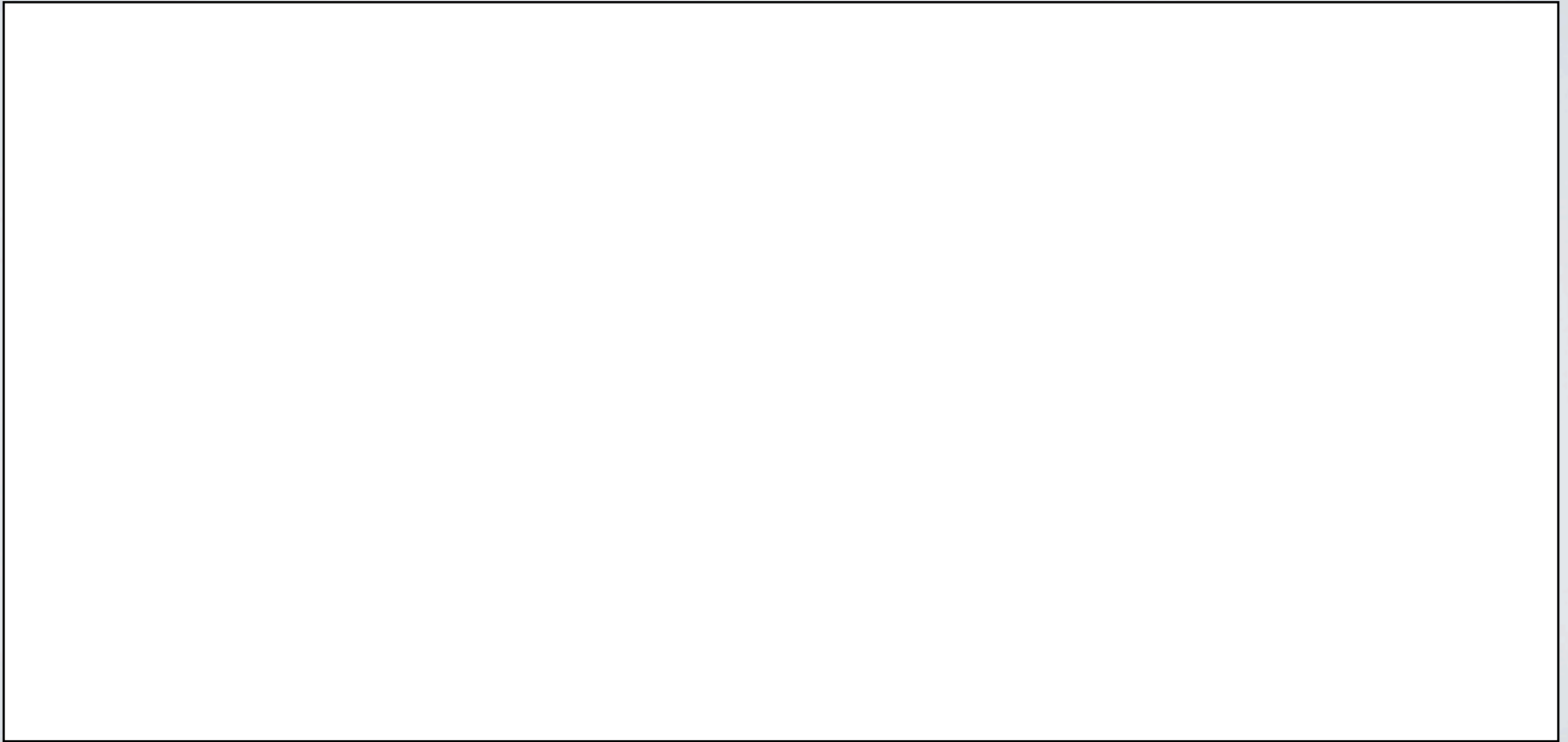
ne sera définie que pour les types dérivés...

# C RTP

les entiers sont affichables...



# CRTP



# C RTP

- 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

# C RTP

- 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*)

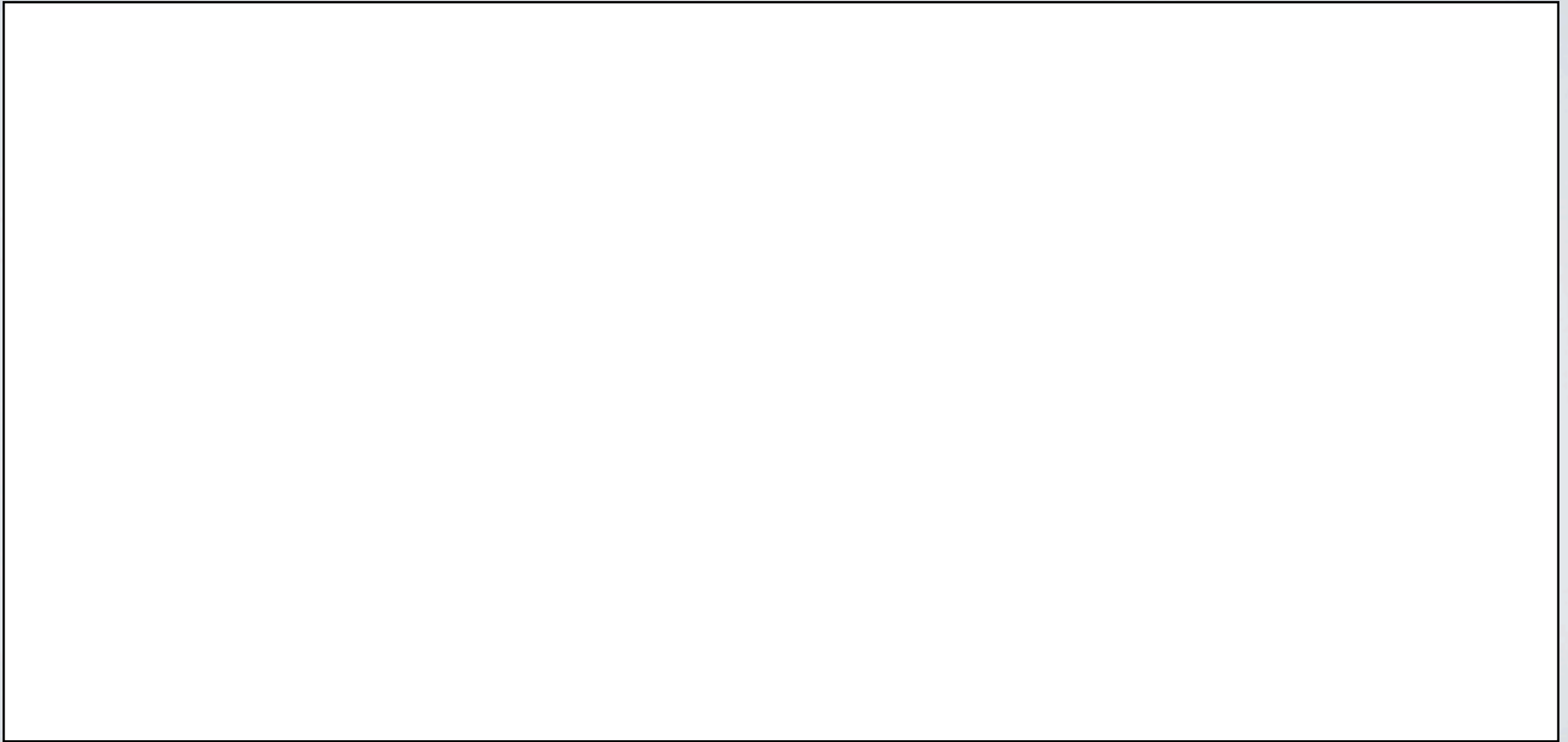
# CRTP

`equivalence<T>` définit une relation d'équivalence pour `T` si `T::operator<(const T&) const` existe

# C RTP

un nombre est conforme à la relation d'équivalence

# CRTP



# C RTP

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

# C RTP

on crée une classe intermédiaire qui implémente clone pour toutes les sous-classes...



# C RTP

désormais les sous-classes « héritent » d'une méthode de clonage générique qui renvoie le bon clone pour chaque sous-classe