

LE LANGAGE C++ MASTER I UN C ÉTENDU ?

Jean-Baptiste.Yunes@univ-paris-diderot.fr
U.F.R. d'Informatique
Université Paris Diderot - Paris 7

2013—2014

LA COMPILATION

un compilateur polymorphe...

- permet de compiler plusieurs langages (sauf exception) : C, C++, Fortran, et
- supporte les standards C89, C99, et
- est la base du dernier standard :

Convention de nommage du code source :

- un code source C++ prend place dans un fichier d'extension
- un fichier d'entête (ne contenant que des déclarations sauf exception *cf.* templates) est d'extension



La compilation s'effectue en invoquant le compilateur sur le fichier source :

fichier

cela génère un fichier objet

fichier

l'édition de lien peut être faite en invoquant le compilateur :

fichierExécutable

ces étapes peuvent être agrémentées d'options...

LES MOTS RÉSERVÉS

Les **mots réservés** du C++

alignas alignof asm auto bool break case catch char
char16_t char32_t class const const_cast constexpr
continue decltype default delete do double
dynamic_cast else enum explicit export extern false
float for friend goto if inline int long mutable
namespace new noexcept nullptr operator private
protected public register reinterpret_cast return
short signed sizeof static static_assert static_cast
struct switch template this thread_local throw true
try typedef typeid typename union unsigned using
virtual void volatile wchar_t while

Les **mots réservés** du C++ hérités du C

auto break case char const continue default
do double else enum extern float for goto
if inline int long register return short
signed sizeof static struct switch typedef
union unsigned void volatile while

- leur sémantique est conservée

Les 41 **mots réservés** propres au C++

alignas alignof asm bool catch char16_t
char32_t class const_cast constexpr
decltype delete dynamic_cast explicit
export false friend mutable namespace new
noexcept nullptr operator private protected
public reinterpret_cast static_assert
static_cast switch template this
thread_local throw true try typeid
typename using virtual wchar_t

Les 11 mots réservés étranges du C++

and and_eq bitand bitor compl not not_eq
or or_eq or or_eq

Ils ne sont pas essentiels mais permettent d'écrire des expressions sans utiliser de caractères *spéciaux*.

```
if (x not_eq y) { /* ils sont différents! */ }
```

Divers

alignas alignof asm bool catch char16_t
char32_t class const_cast constexpr
decltype delete dynamic_cast explicit
export false friend mutable namespace
new noexcept nullptr operator private
protected public reinterpret_cast
static_assert static_cast template this
thread_local typeid
typename wchar_t

Booléens

alignas alignof asm **bool** catch char16_t
char32_t class const_cast constexpr
decltype delete dynamic_cast explicit
export **false** friend mutable namespace
new noexcept nullptr operator private
protected public reinterpret_cast
static_assert static_cast template this
thread_local throw **true** try typeid
typename using virtual wchar_t

Exceptions

alignas alignof asm bool **catch** char16_t
char32_t class const_cast constexpr
decltype delete dynamic_cast explicit
export false friend mutable namespace
new **noexcept** nullptr operator private
protected public reinterpret_cast

throw

try

Objets

alignas alignof asm bool catch char16_t
char32_t **class** const_cast constexpr
decltype delete dynamic_cast **explicit**
export false friend mutable namespace
new noexcept nullptr operator private
protected public reinterpret_cast
static_assert static_cast template **this**
thread_local throw true try typeid
typename using **virtual** wchar_t

Conversions

alignas alignof asm bool catch char16_t
char32_t class **const_cast** constexpr
decltype delete **dynamic_cast** explicit
export false friend mutable namespace
new noexcept nullptr operator private
~~protected public~~ **reinterpret_cast**
static_cast

Espaces de noms

namespace

using

Généricité

alignas alignof asm bool catch char16_t
char32_t class const_cast constexpr
decltype delete dynamic_cast explicit
export false friend mutable namespace
new noexcept nullptr operator private



template

LES OPÉRATEURS

- Les opérateurs du C++ (priorité I) :

	portée	
--	--------	--



- Les opérateurs du C++ (priorité 2) :

	appel de fonction	
	initialisation membres	
	indice tableau	
	accès par pointeur	
	accès par objet	
	post-incrémentation	
	post-décrémentation	
	coercition	
	coercition	
	coercition	
	coertition	
	RTTI	


- Les opérateurs du C++ (priorité 3) :

ou	non logique	
ou	complément b.àb.	
	pré-incrémentation	
	pré-décrémentation	
	moins unaire	
	plus unaire	
	déréférencement	
	adresse	
	allocation	
	allocation tableau	
	désallocation	
	désallocation tableau	
<i>T</i>	coercition	
	taille représentation	

- Les opérateurs du C++ (priorité 4) :

	sélecteur (pointeur)	
	sélecteur (objet)	

- Les opérateurs du C++ (priorité 5) :

	multiplication	
	division	
	reste modulo	

- Les opérateurs du C++ (priorité 6) :

	addition	
	soustraction	

- Les opérateurs du C++ (priorité 7) :

	décalage gauche b.àb.	
	décalage droit b.àb.	

- Les opérateurs du C++ (priorité 8) :

	inférieur strict	
	inférieur	
	supérieur strict	
	supérieur	

- Les opérateurs du C++ (priorité 9) :

ou	comparaison égalité	
ou	comparaison différence	

- Les opérateurs du C++ (priorité 10) :

ou	et b.àb.	
----	----------	--

- Les opérateurs du C++ (priorité I I) :

ou

ou-exclusif b.àb.

- Les opérateurs du C++ (priorité 12) :

ou	ou b.àb.	
----	----------	--

- Les opérateurs du C++ (priorité 13) :

lazy evaluation

ou

et logique

- Les opérateurs du C++ (priorité 14) :

lazy evaluation

ou	ou logique	
----	------------	--

- Les opérateurs du C++ (priorité 15) :

	expression conditionnelle	
--	------------------------------	--

- Les opérateurs du C++ (priorité 16) :

	affectation	
	incrémentation et affectation	
	décrémentation et affectation	
	multiplication et affectation	
	division et affectation	
	reste modulo et affectation	
ou	et b.àb. et affectation	
ou	ou-exclusif b.àb. et affectation	
ou	ou b.àb. et affectation	
	décalage gauche et affectation	
	décalage droit et affectation	

- Les opérateurs du C++ (priorité 17) :

	levée d'exception	
--	-------------------	--

- Les opérateurs du C++ (priorité 18) :

	évaluation séquentielle	
--	-------------------------	--



- Les priorités permettent d'interpréter les expressions multi-opérateurs :
- Certains opérateurs sont gauches :

- Les commentaires :
 - multiligne

```
/*  
    cette suite d'instructions est la plus  
    importante du programme  
*/  
i = i+1;  
j = i-1;
```

- uniligne

```
i = i+1; // et un de plus!
```



LES VARIABLES

- Quelques rappels sur les variables :
- Une variable est un contenant informatique nommé sur lequel des opérations de consultation et modification du contenu sont définies
 - la *lecture* produit une *r-value*
 - l'écriture se fait par l'intermédiaire de la *l-value*
- Le nom est appelé identificateur de variable.
En on parle aussi de **référence**

- La déclaration d'une variable :
 - consiste en la création d'une référence
 - portée/visibilité
 - nécessite un type
 - permet de contrôler l'usage

=

+

- La définition d'une variable est :

- est une déclaration (agrémentée ou non d'un qualifieur : `auto`, `static`)
- associée à une allocation mémoire
- le tout éventuellement suivi d'une initialisation

- Les usages d'une variable :
 - la consultation s'effectue en utilisant en *r-value* la référence
 - la modification s'effectue en utilisant en *l-value* la référence : instruction d'affectation
- Attention : **initialisation et affectation sont deux opérations distinctes** (il ne peut jamais y avoir qu'une seule initialisation pour une variable donnée!)

- En :

- une déclaration :

- une définition :

- une définition avec initialisation :

- une affectation :

- Attention :

Pour les curieux est un opérateur surchargé, s'utilise pour exprimer différentes choses...

- En :

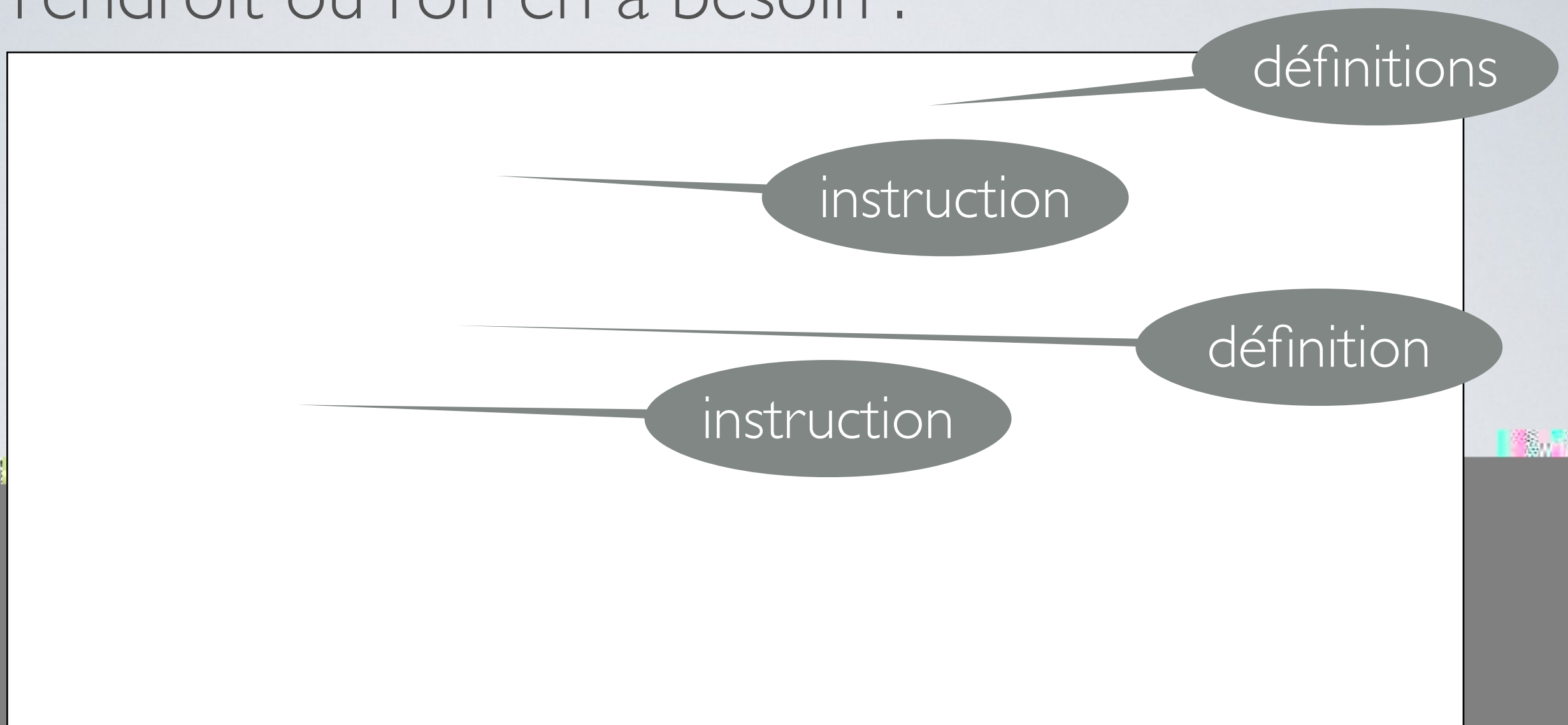
- syntaxe fonctionnelle pour l'initialisation :

qui s'interprète exactement comme

- peut être utilisé pour n'importe quel type (on verra plus tard d'autres cas intéressants) :



- En :
- il est possible de déclarer/définir des variables à l'endroit où l'on en a besoin :



- pour **for** : la définition a la portée de la boucle...
Attention c'est faux pour les compilateurs Microsoft!

LES VARIABLES CONSTANTES

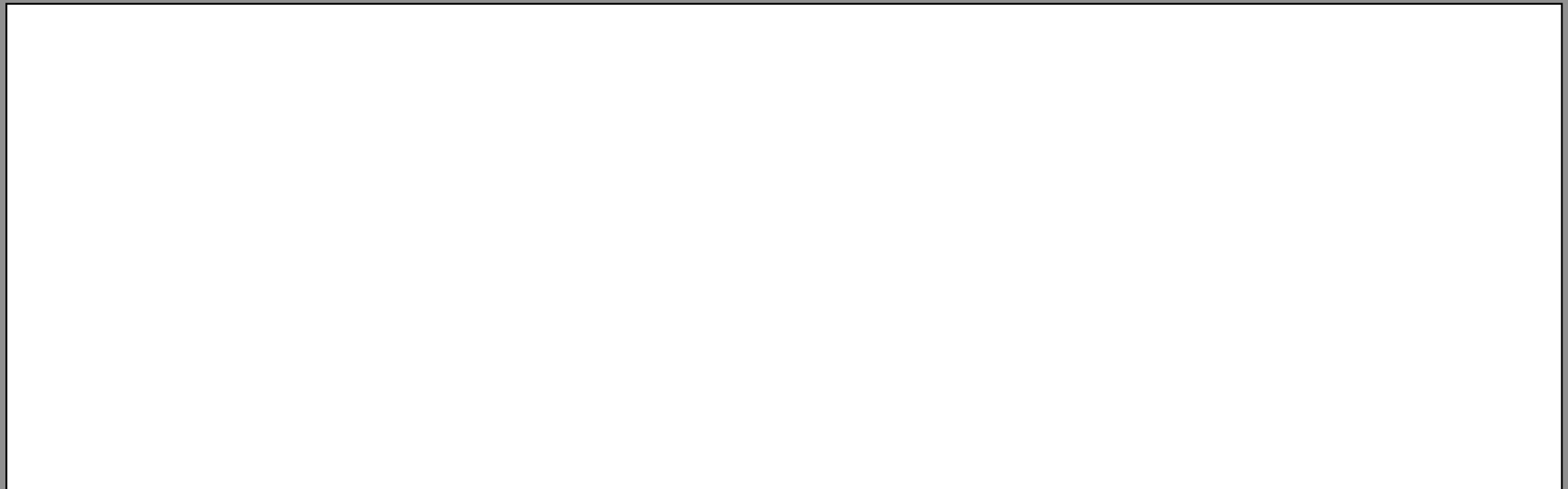
- Les types constants : *type*
 - interdit l'usage en tant que *l-value*, i.e. les opérateurs qui modifient la valeur de la variable du type considéré (ex: ,)
 - initialisation obligatoire
 - le reste est identique au type non-constant



- Dans les prototypes :

- on a la garantie que les caractères de la chaîne source ne seront pas modifiés lors d'un appel à la fonction
- celui qui implémente la fonction ne peut modifier les caractères de l'argument constant

- En :
 - les constantes sont des variables (contraintes car non modifiables) qui peuvent être employées partout où des littéraux constants peuvent l'être :



Recommandation : éviter d'employer les **#define** du pour définir des constantes



L'ALLOCATION DYNAMIQUE

L'allocation dynamique en C++ **new** :

type

type valeur initiale

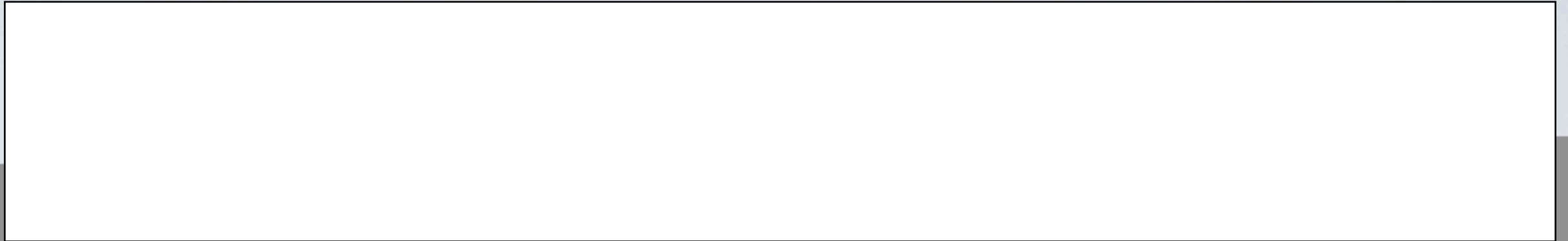
type dimension

- pas d'initialisation de tableaux à l'allocation dynamique

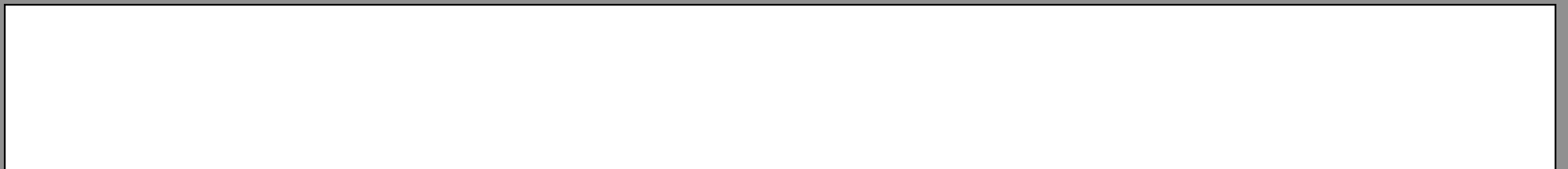
Attention : ne pas employer `new` et `delete` / `new` et `delete[]` en même temps. On fait du `new` ou du `delete`.

L'allocation dynamique en C++ `delete` :

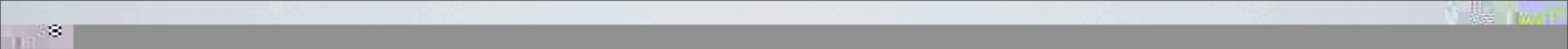
adresse



adresse



Attention : ne pas employer `delete` et `delete[]` en même temps. On fait du `delete` ou du `delete[]`.



LE PROTOTYPAGE

Le **prototypage** en C++

- obligatoire! est bien plus strict que le
- déclaration d'une fonction appelée , prenant 3 arguments respectivement un entier, un flottant et un entier et renvoyant une valeur entière :

- déclaration d'une fonction ne prenant PAS d'arguments (et renvoyant un entier) :

attention car en c'est car
signifie fonction à nombre d'arguments indéfini!

Les **valeurs par défaut** en argument (sont à placer dans les déclarations)

```
const int RADIAN=0;
const int DEGRE=1;
const int GRADE=2;

double sinus(double angle,int unite=RADIAN);

void main() {
    double angle=3.14, s;
    s = sinus(angle);
    double angleEnDegres = 90, sd;
    sd = sinus(angleEnDegres,DEGRE);
}
```

Les **valeurs par défaut** en argument

- un polymorphisme très pauvre
- ne peuvent être arbitrairement mélangées. La liste des arguments d'une fonction est sécable en deux parties (éventuellement vides)
 - d'abord les arguments sans valeur par défaut. À l'appel il doivent tous être spécifiés
 - ensuite ceux avec valeur par défaut. À l'appel on peut spécifier des valeurs pour les premiers d'entre eux et laisser le compilateur compléter la spécification à l'aide des valeurs par défaut

La glu C/C++

- pour déclarer une fonction C à utiliser dans le monde C++, il faut la déclarer comme telle (les transmissions d'arguments peuvent être différents d'un monde à l'autre) :



LA SURCHARGE

La **surcharge** de fonctions

- Comment résoudre élégamment le problème suivant :
- Écrire une fonction permettant d'additionner deux entiers, puis une autre deux flottants...

En :

mais il faut de la mémoire ou une bonne documentation....

La **surcharge** de fonctions

- Écrire une fonction permettant d'additionner deux entiers, puis une autre deux flottants. En

pas de problème de conflit de nom, le compilateur est suffisamment malin pour deviner quoi faire avec :

Attention : pas de prise en compte de la valeur de retour.

Signature = Nom + Liste des Types

La directive :

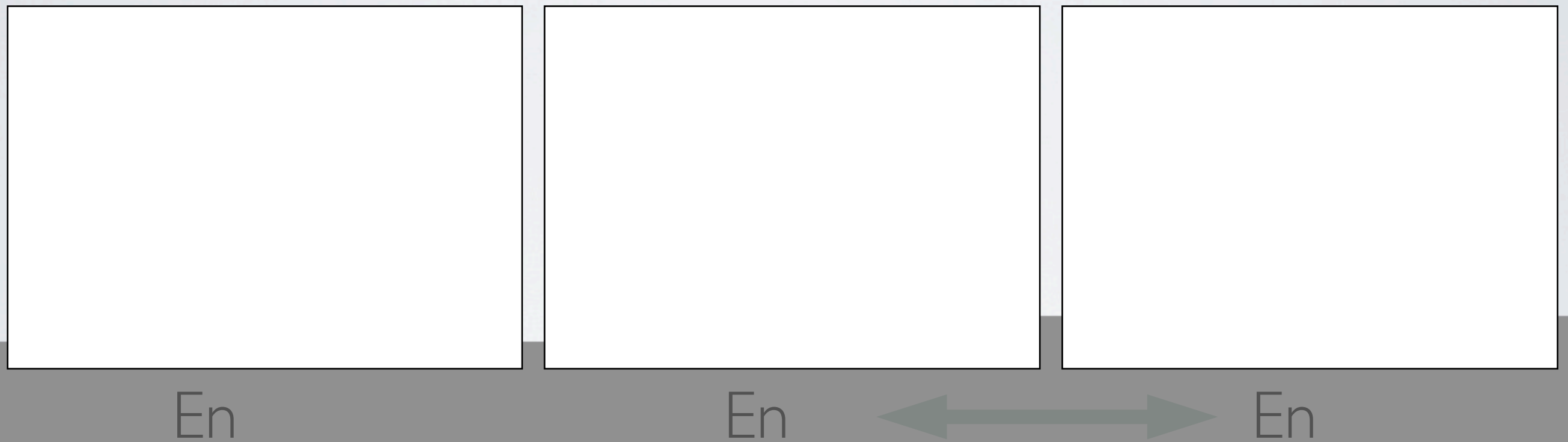
- permet d'économiser l'appel de fonction... mais peut être ignorée par le compilateur... donc inutile d'y prêter attention...

même rôle que la macro-définition mais avec vérification des types et sans effet secondaire.

Attention : la fonction est considérée comme
il est donc impossible d'en prendre
l'adresse (sinon est ignoré)

PORTABILITÉ C/C++

- Incompatibilités / :
- types structurés. En , *type* est le nom du type défini, en *type* est le nom du type définit par *type*
- en les types imbriqués le sont vraiment! En , ils sont aplatis :

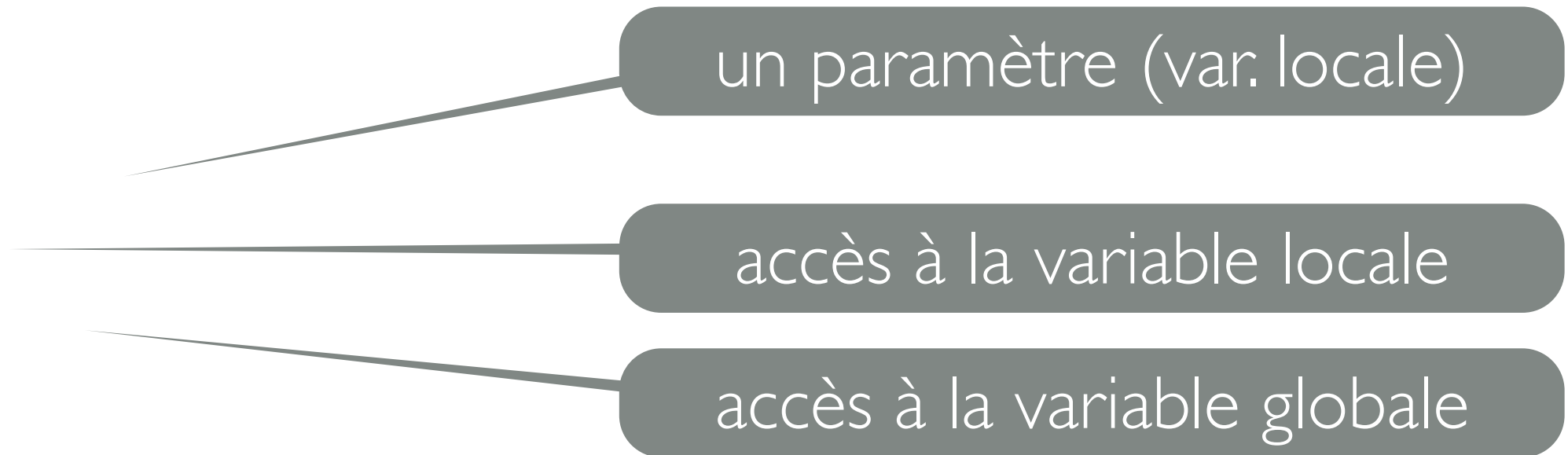


Utilisation de l'opérateur



LES PORTÉES DES NOMS

- L'opérateur pour contrecarrer l'effet du masquage :



En réalité l'opérateur permet d'accéder à un nom dans un contexte donné (espace de noms)

pas de contexte = contexte global

- Les espaces de noms en :
- les conflits de noms sont inévitables mais comment limiter leur impact ?

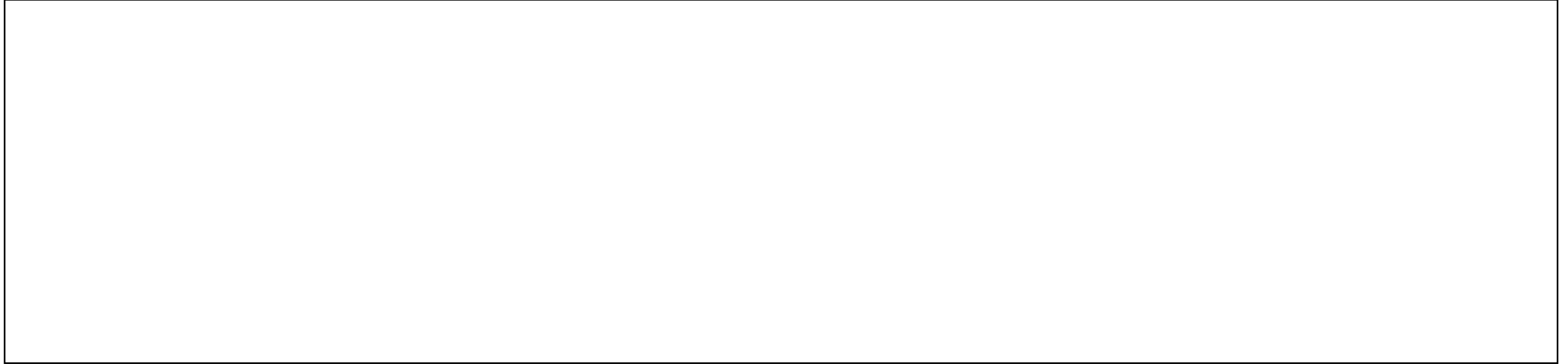
En encapsulant les déclarations dans des espaces...

maths.hpp

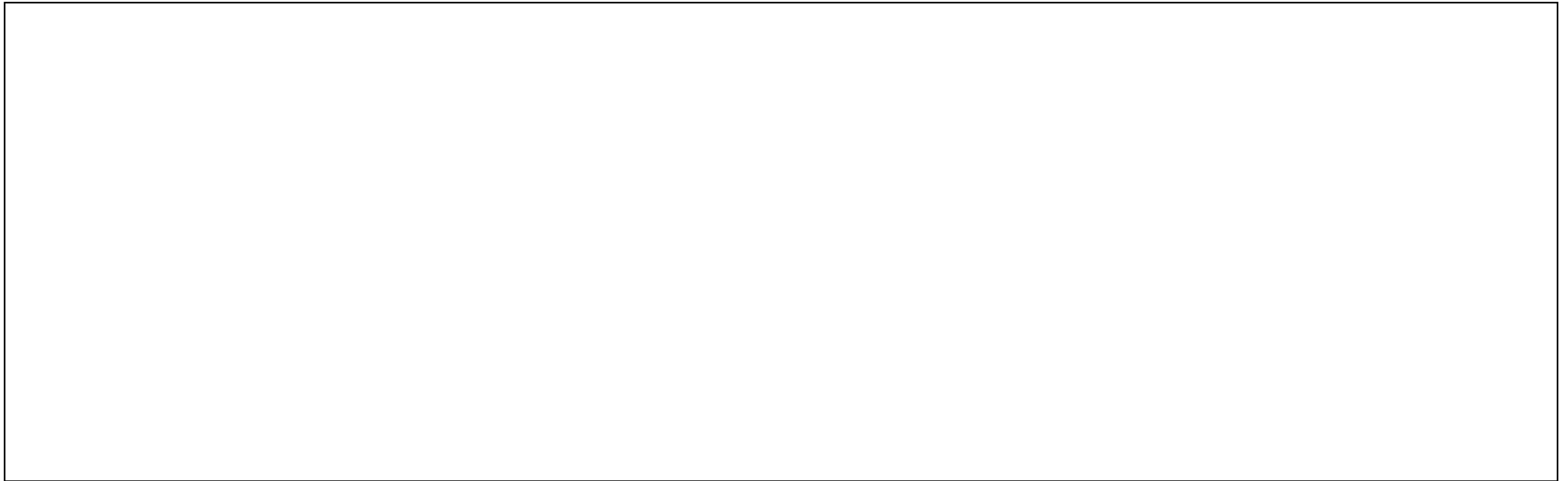
chimie.hpp

ailleurs.cpp

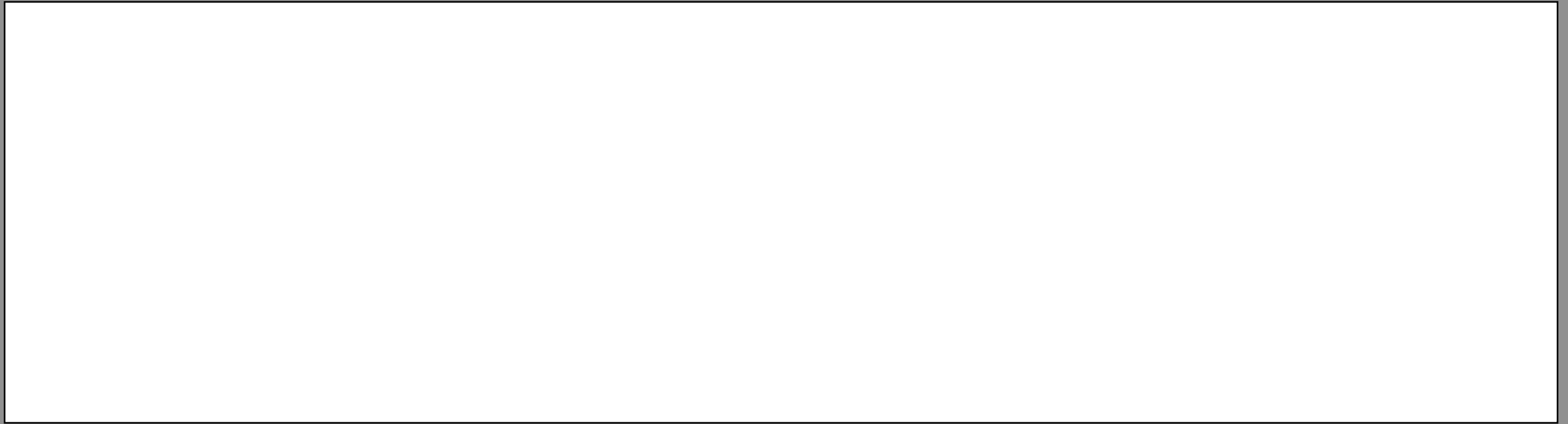
- Usage du `using` via l'opérateur de portée `::` :



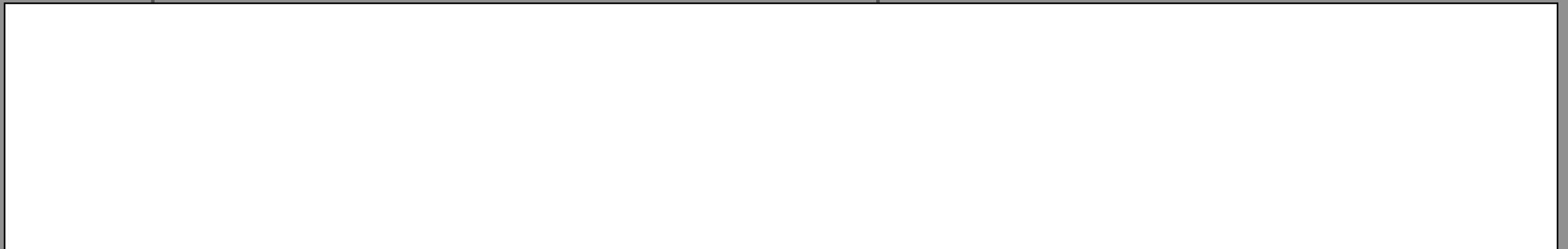
- Le compilateur retrouve ses petits s'il n'y a pas d'ambiguïté via la mise à plat des espaces de noms :



- Il existe un espace de noms (par défaut) sans nom
- on peut imbriquer les espaces de noms



- on peut définir des alias d'espace de noms



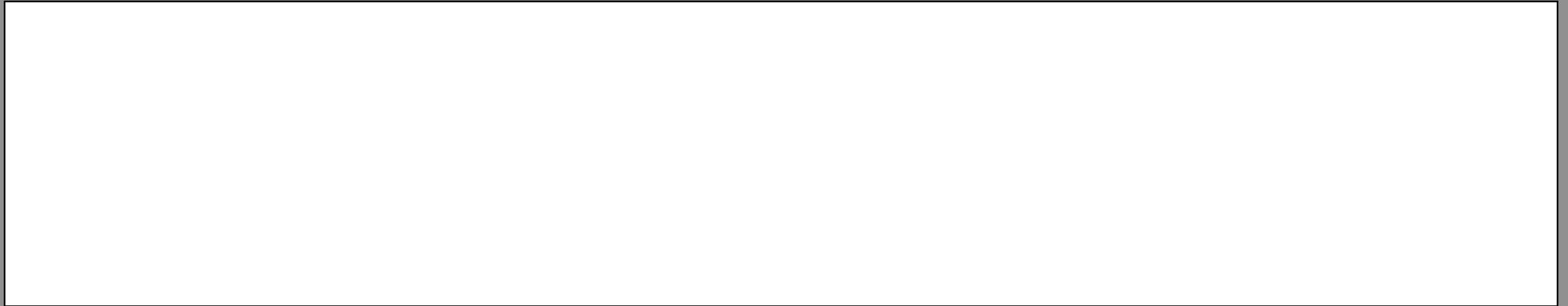


LES RÉFÉRENCES

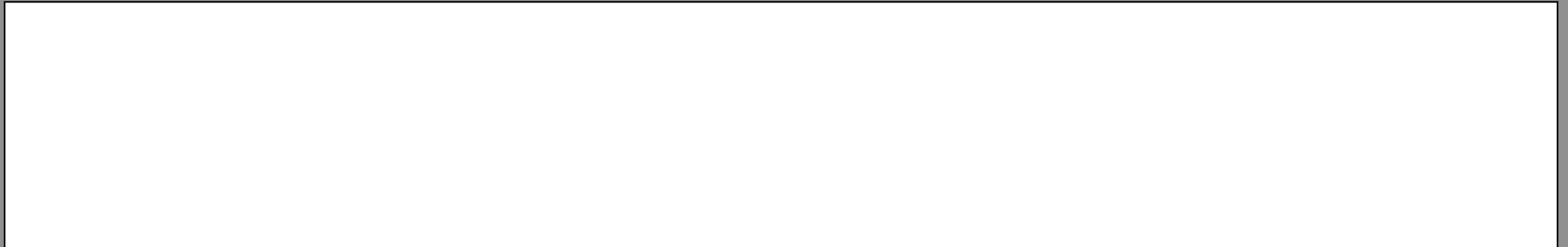
- Les types références
 - On rappelle ici que **la sémantique, du c,** pour le **passage d'arguments** à l'appel de fonctions est **par valeur**. C'est pourquoi on ne peut écrire de fonction permettant d'échanger la valeur de deux variables. L'*astuce* est alors de passer les arguments par l'intermédiaire de leur adresse

Attention : il s'agit encore d'un passage par valeur : on passe la valeur de pointeurs...

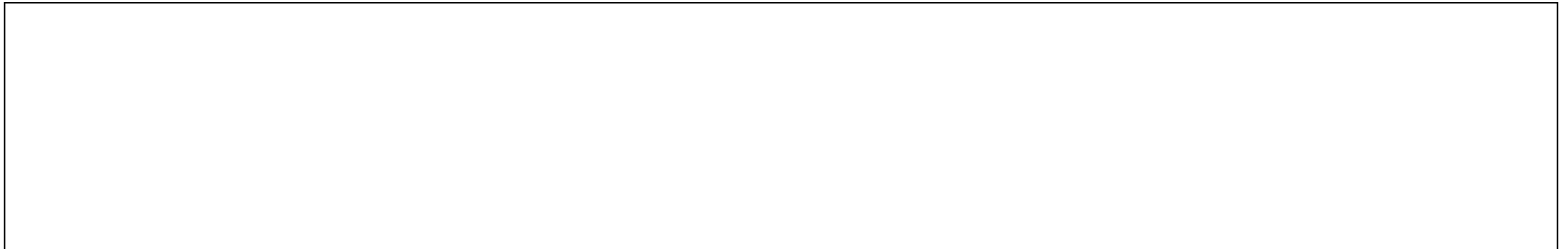
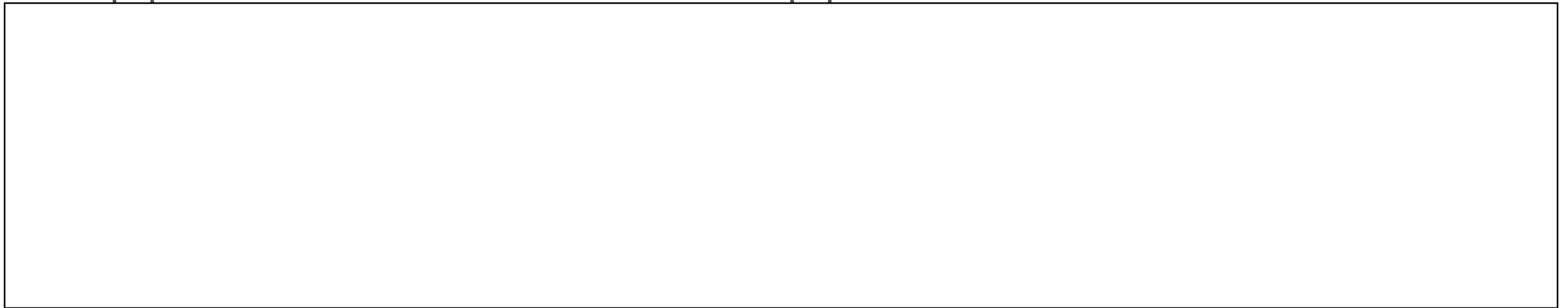
- Les types références :




- Nécessite l'emploi explicite des pointeurs...



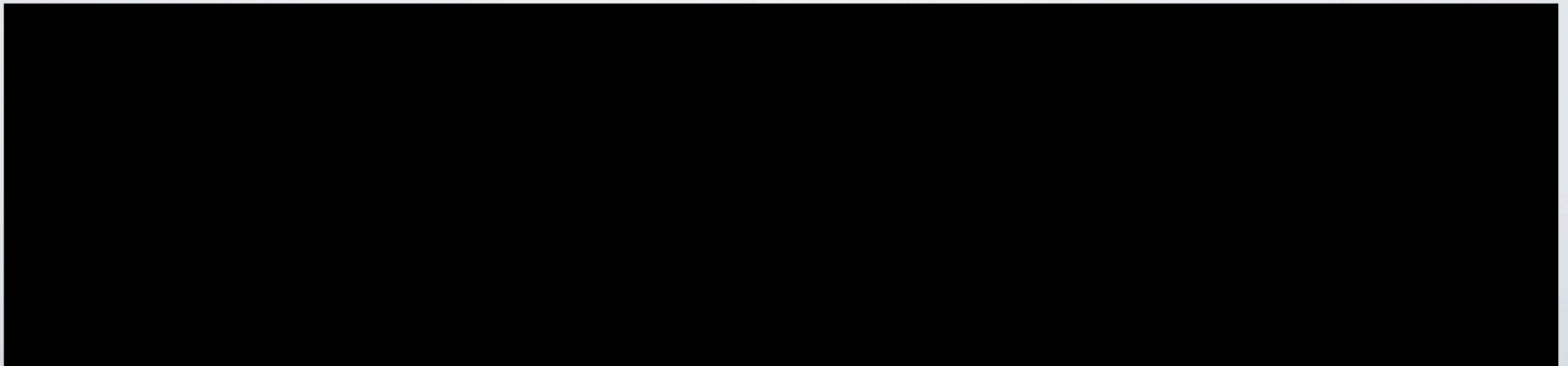
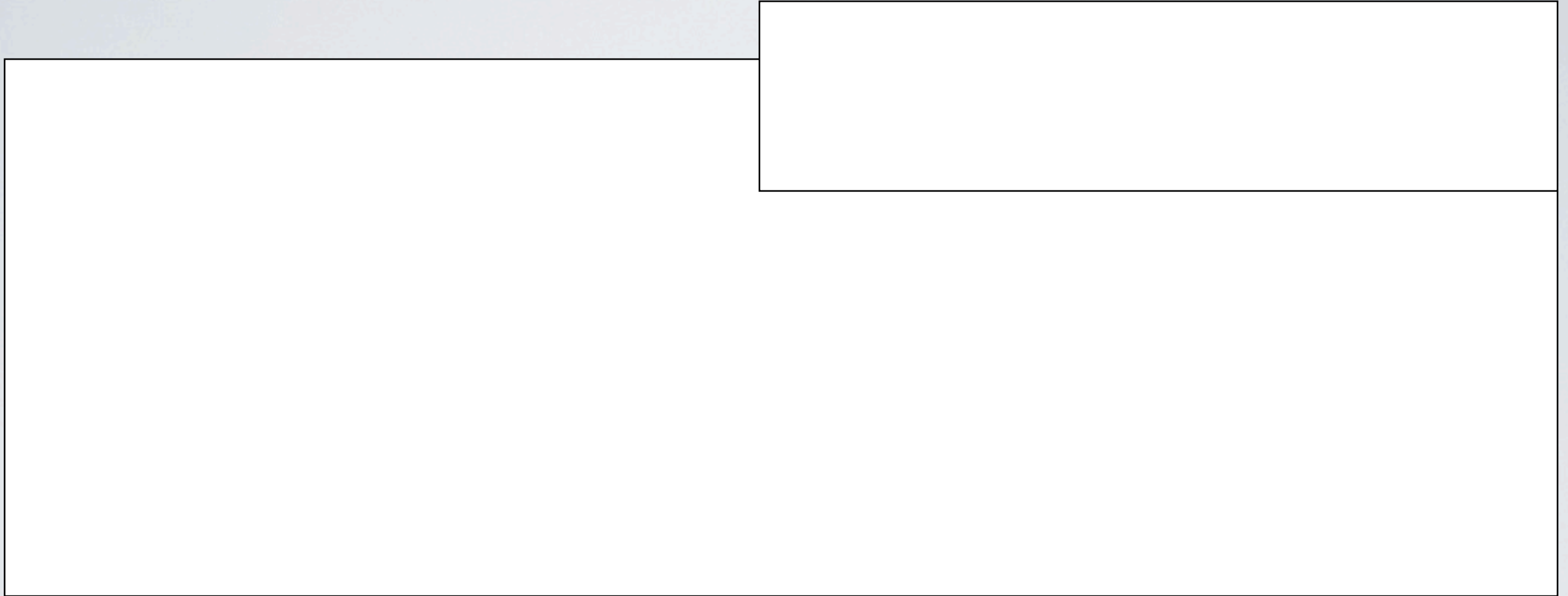
- Les types références :
- On peut désormais changer la sémantique de l'appel sans modifier les appels eux-mêmes



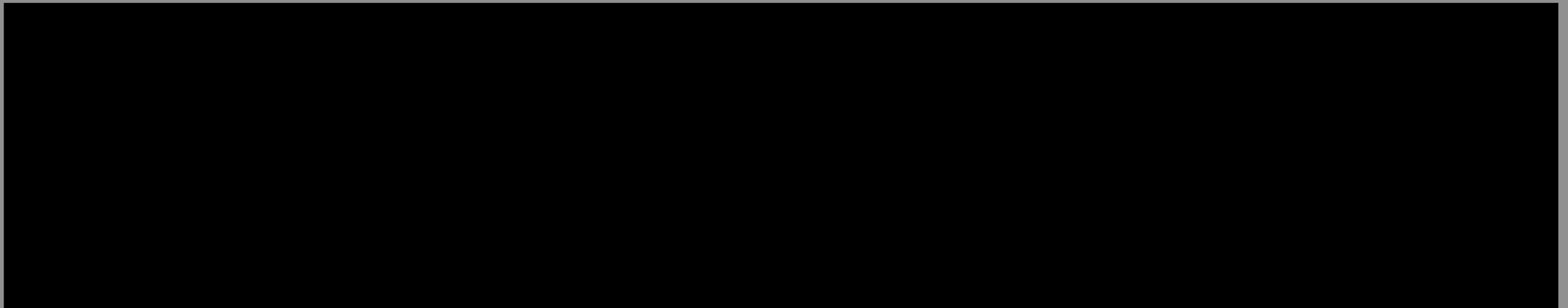
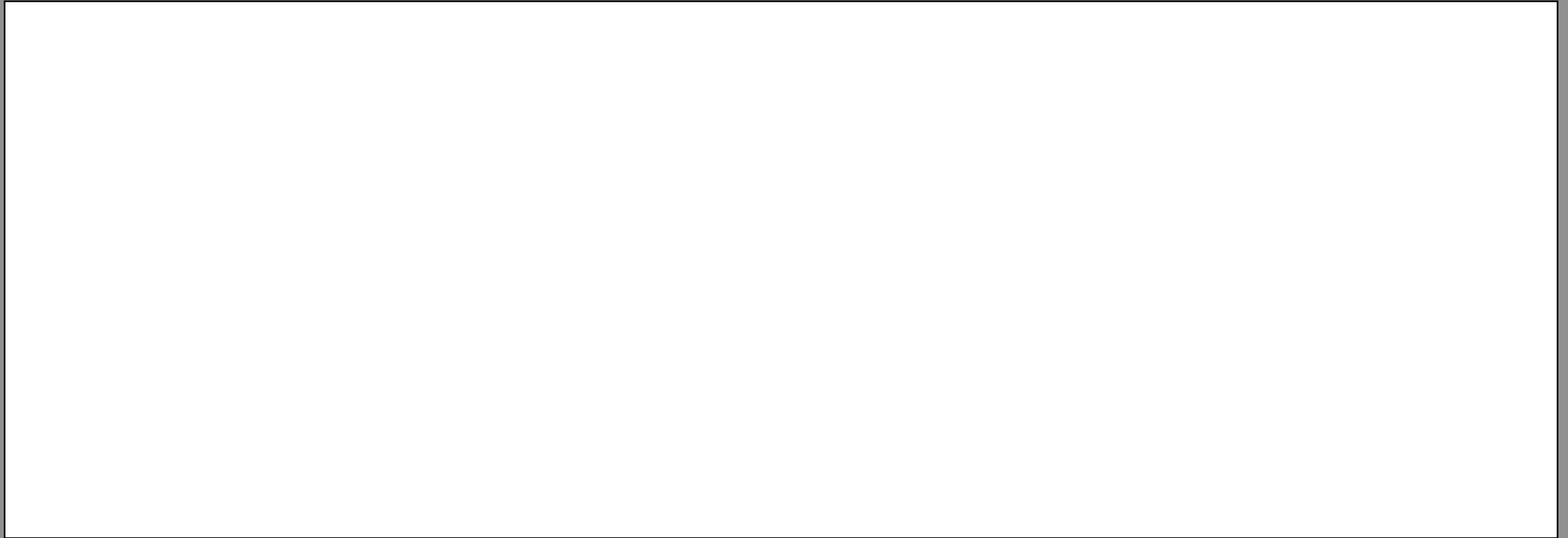
- Pas de création de nouvelle variable, création d'une référence, *i.e.* un alias du paramètre effectif

- Les types références :
- L'utilisateur aimerait parfois obtenir la garantie que ces arguments ne sont pas modifiés, d'où l'emploi de  et qui permet aussi de passer des constantes ou des littéraux :

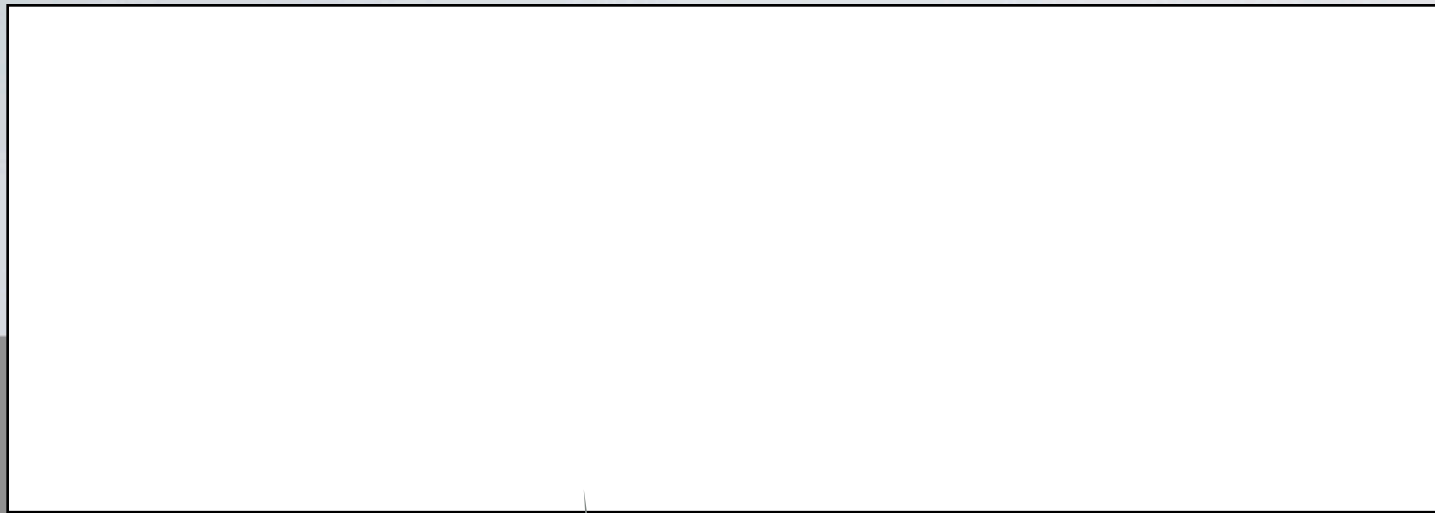
- Les types références (plus bizarre !?) :
- On peut aussi renvoyer des références, une :



- Les types références (plus rare !?) :
- On peut déclarer des références, la condition nécessaire est de les initialiser...

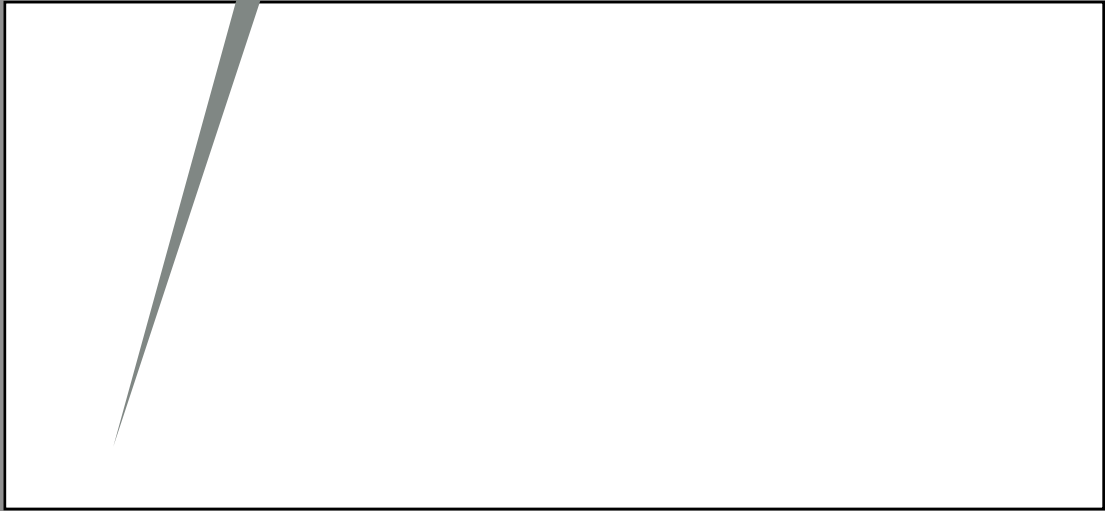


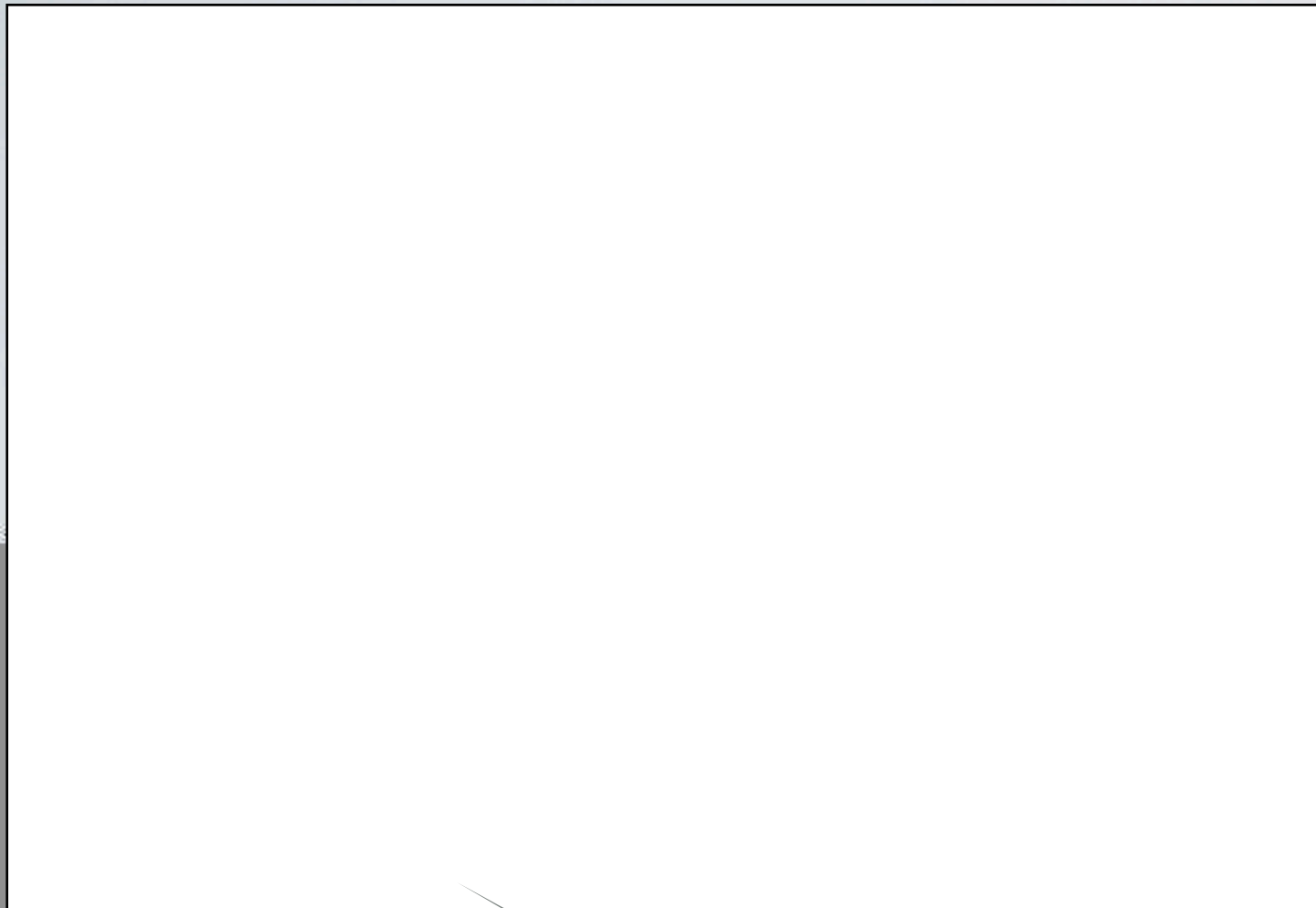
INCOMPATIBILITÉS ?



en le type est
Personne

en le type est
struct Personne





en les types sont
vraiment imbriqués