Les structures en C

Nicolas Gazères

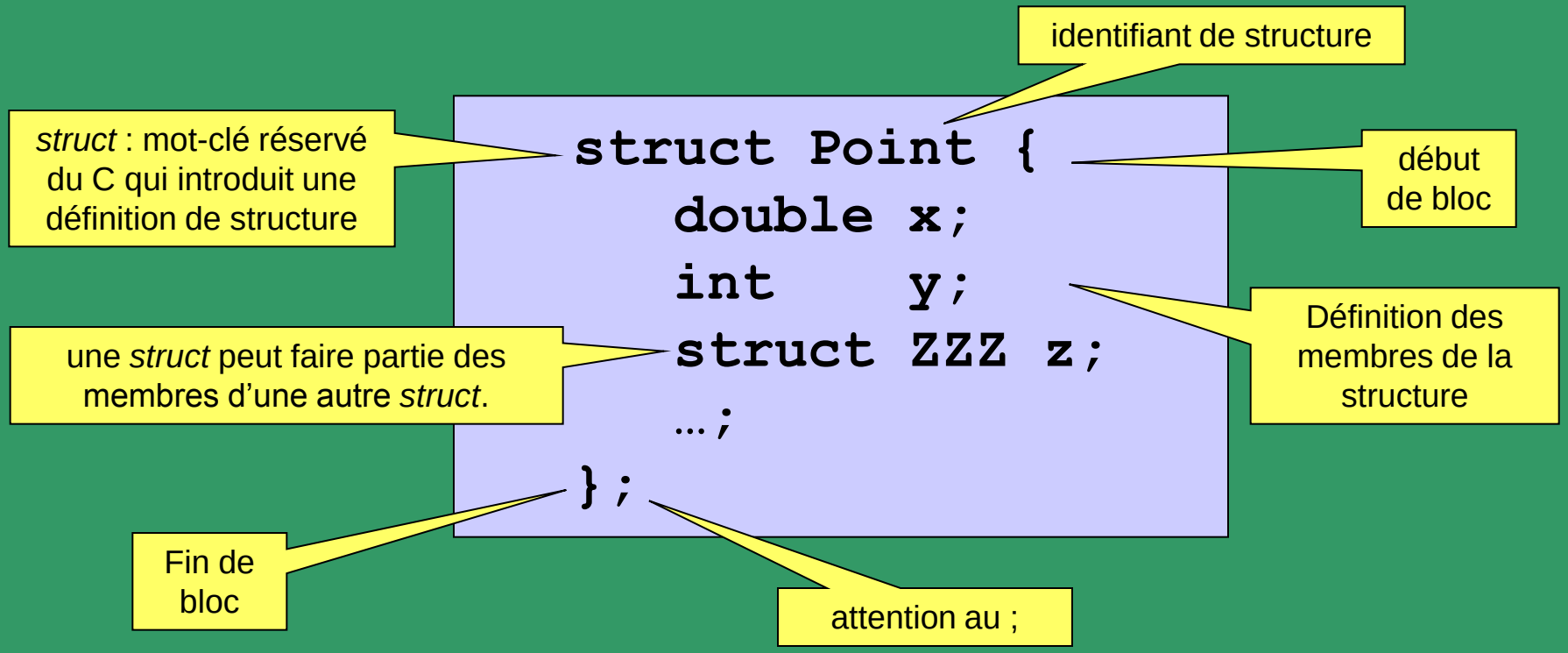
Dassault Systèmes
DS Research, Life Sciences
ngs@3ds.com

Finalité des structures

- Construire des types (ie. des ensembles)
  - qui expriment dans la modélisation informatique les concepts du domaine d'étude
    - « ***Types utilisateurs*** »
  - ***Par composition***
    - à partir des types standards (*int*, *double*,…), ou
    - à partir d'autres types utilisateurs « plus élémentaires »
- Les structures sont le formalisme que le langage C fournit pour élever le **niveau d'abstraction** des modèles.
  - Sémantique et complexité croissantes
- Ainsi, on peut construire:
  - une hiérarchie de types
  - une échelle d'abstraction croissante

Définition d'un type structure
Syntaxe

- La « *struct* » est le mécanisme offert par le langage C pour manipuler des concepts plus abstraits.
- On traite des données liées entre elles de manière *solidaire* plutôt que de les traiter indépendamment.
- La définition de la *struct* est en général placée dans un *header.*



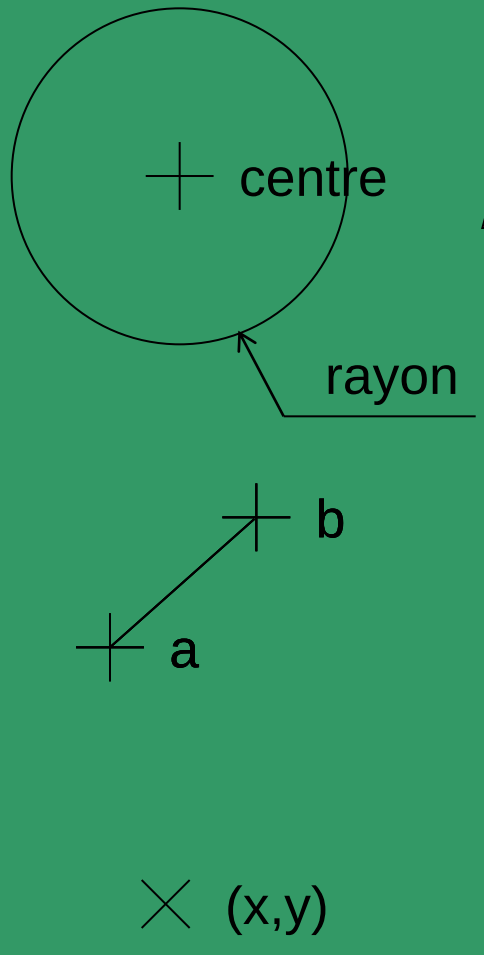

```
struct Point {
    double x;
    int    y;
    struct ZZZ z;
    …;
};
```

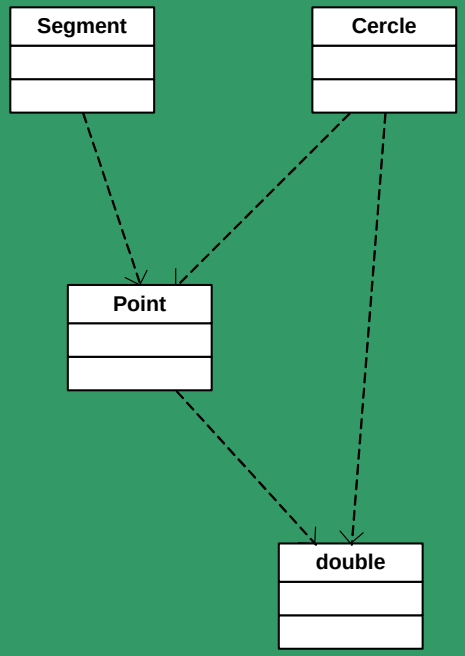
Exemple

```
struct Cercle {
  struct Point centre;
  double rayon;
};
```

```
struct Segment {
  struct Point a;
  struct Point b;
};
```

```
struct Point {
  double x;
  double y;
};
```





Déclaration d'une « *instance de struct* »

- Le mot-clé *struct* introduit un nouveau type à part entière
  - C'est-à-dire un ensemble *d'instances* ayant des propriétés communes.
  - Ce type est utilisable exactement comme les types standards (*int*, *float*, …)
- Syntaxe

Point.h

```
struct Point {
  double x;
  double y;
};
```

main.cpp

```
int main() {
  struct Point P1, P2;
  …
}
```

Réserve de la mémoire pour deux instances

- On peut créer autant d'instances de la *struct* que la mémoire le permet:
  - toutes les instances partagent la même « structure » (rigide)
  - mais chaque instance possède son *état interne propre*
    - *La valeur de ses données-membres*
  - analogie *moule/pièce moulée*.

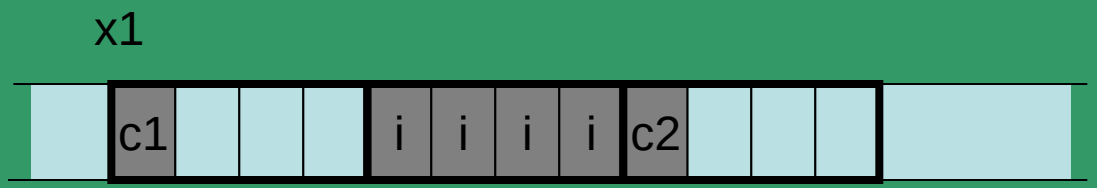
Occupation mémoire des *structs*

Alignement des champs

- Sur les machines récentes, il existe des contraintes d'*alignement*.
  - chaque variable d'un type natif est aligné sur une adresse multiple de son « modulo d'alignement » (*char*=1, *int*=4, *double*=8,…)
  - Les membres des *structs* n'échappent pas à la règle.

- Conséquences
  - Toute *struct* a son propre « modulo d'alignement ».
    - Le modulo d'une *struct* est toujours *au plus égal* au modulo de type natif le plus élevé (en général, *double*)
  - Il peut y avoir des « trous » dans l'organisation interne d'une *struct*:

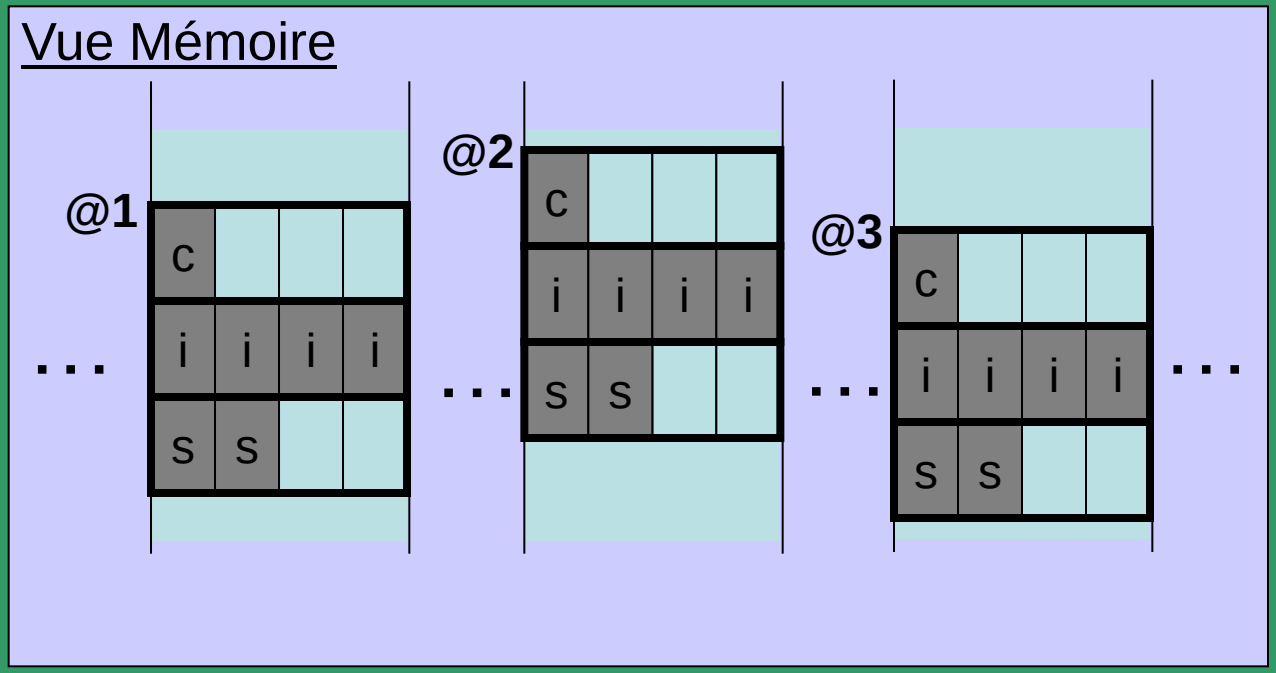
```
struct X {
   char  c1 ;
   int  i;
   char  c2;
};
```



Rigidité des structures

- La disposition interne d'une *struct* est « rigide »
  - elle est reproduite à l'identique pour chaque instance
    - pas de déformation à l'intérieur d'une structure
- Intérêt
  - Le compilateur accède à un membre donné en utilisant le *même offset* (décalage fixe), quelle que soit l'instance de *struct*.

```
struct X {
   char  c;
   int   i;
   short s;
};
```



Sizeof
Cas des structs

- La valeur du *sizeof* est définie comme:
  - le décalage en octets entre deux éléments consécutifs d'un tableau de *struct*. (deux éléments consécutifs ne seront pas tassés, cf. *rigidité*)
  - de manière équivalente: le décalage en octets réalisé par l'opérateur ++ appliqué à un pointeur vers struct.

- Exemple

```
struct X {
   char  c1 ;
   int   i;
   char  c2;
};
```

- Que donne *sizeof(struct X) ?*
  - Valeur attendue → **6** (1+4+1)
  - Valeur observée → **12**

- L'opérateur *sizeof* donne la taille en octets d'une *struct*, espaces internes d'alignement compris.

Occupation mémoire des *structs*

Recommandation

- Dans une définition de *struct*, ordonner les membres par taille croissante (ou décroissante), mais éviter les extrema.

```
struct X {
   char  c1 ;
   char  c2;
   int  i;
};
```

OK

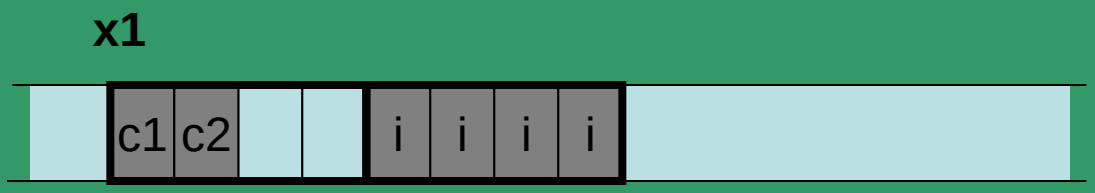
```
struct X {
   char  c1 ;
   int  i;
   char  c2;
};
```

inefficace

```
struct X {
   int  i;
   char  c1 ;
   char  c2;
};
```

OK

- Nouvelle disposition (optimisée en espace)
  - *Charge utile* → 6 (1+1+4)
  - *sizeof* obtenu → 8

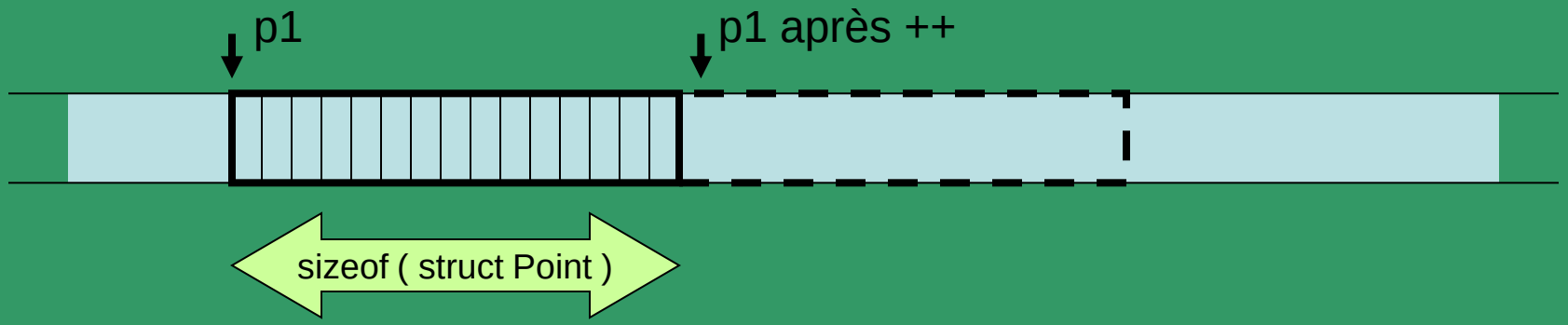


Pointeurs et tableaux de *struct*

- Chaque instance de *struct* possède une ***adresse-mémoire*** propre, que l'on peut récupérer par l'opérateur d'adressage (&).
- L'adresse de l'instance de *struct* peut être manipulée via un pointeur de type « pointeur sur *struct Point* » :

```
struct Point c1;
struct Point *p1 = &c1 ;
```

- L'opérateur ++ fonctionne sur les « pointeurs d'instance de *struct* ».
  - p+1 décale le pointeur à l'instance de *struct* suivante.
  - En octets, le décalage correspond au *sizeof* de la structure.



Allocation dynamique

- Une instance de structure peut être allouée **dynamiquement**.
  - Sur le tas (« *heap* »)
- Comme pour les autres types, l'**allocation** dynamique de mémoire pour les *struct* en C se fait par la fonction standard *malloc()*
  - *malloc() détermine et renvoie un pointeur (*typé *void*) sur une zone-mémoire contenant le nombre d'octets demandé (peut-être quelques octets de plus…).*
  - *Le programmeur doit calculer lui-même le nombre d'octets nécessaires.*
    - *Risque d'erreur de programmation (faible, mais présent)…*
  - *malloc() prend en compte les contraintes d'alignement-mémoire.*

- *Exemple*
  - Pour allouer un tableau de *N* instances de *struct Point* :

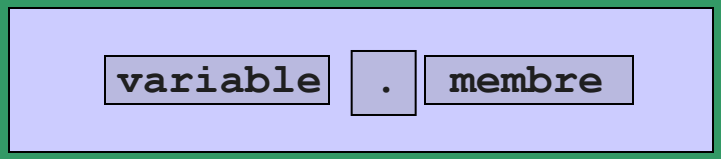
```
struct Point * p1
   = (struct Point*) malloc( N*sizeof(struct Point) );
```

- Comme pour les autres types, la mémoire se désalloue par *free():*

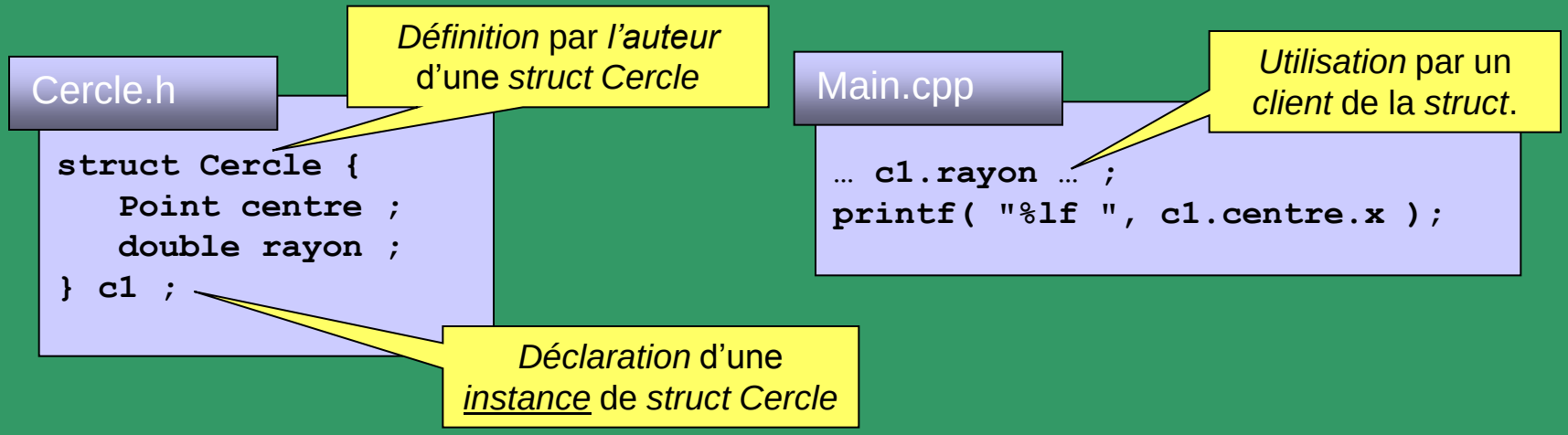
```
free( p1 );
```

Accès aux données d'une *struct* (1)

- L'opérateur « point » (ou *opérateur de sélection*)
  - utilisable sur toute variable d'instance de *struct*.
- Syntaxe



- Exemple
  - L'expression qui désigne la donnée-membre peut être utilisée partout où une expression du même type est attendue.



Accès aux données d'une *struct* (2)

- L'opérateur « **flèche** »
  - utilisable sur toute variable de type « pointeur sur instance de *struct* ».
- Syntaxe : « pointeur -> donnée »
  - la flèche est un tiret (-) suivie d'un supérieur (>)
- Exemple

```
struct Cercle *p1 = &c1 ;

… p1->r … ;
… p1->centre.x … ;
```

- On a l'équivalence suivante:

(*p1).r ⇔ p1->r

conseil :
préférez celle-ci.

Opérations admises sur les *struct* (1)
Initialisation

- Principe
  - on donne entre accolades la liste des valeurs initiales de chaque membre.
  - on respecte *l'ordre* et *le type* des données-membres.
  - si la liste d'initialisation comporte moins d'éléments que la définition de la *struct*, les derniers membres sont initialisés par la valeur 0.
- Exemples

```
struct Point P1 = { 1.0, 2.7 };          // OK

struct Segment S1 = { {1.0, 2.7},
                      {-7.1, 0.0}  };  // OK

struct Cercle C1 = { {0.0, 3.0}, 10.0 };      // OK
struct Cercle C2 = { {0.2, 71.0} };           // OK

struct Segment S1 = { P1, P2 };         // KO
```

Opérations admises sur les *struct* (2)
Copie

- Définition
  - la copie de *struct* est définie comme la copie membre à membre de toutes les données de la *struct*.

- Exemple

```
struct Point P1 = { 1.0, 2.7 };
struct Point P2 = P1;
```

- On a l'équivalence suivante:

```
struct Point p2 = p1;
```

Forme compacte
(copie solidaire)
→ *abstraction plus élevée*

⇔

```
struct Point P2 ;
P2.x = P1.x ;
P2.y = P1.y ;
```

Forme développée
équivalente

Opérations admises sur les *struct* (3)

Passage en paramètre de fonction

- Exemple

Passage de *struct* en arguments d'entrée.

La fonction

```
struct Point milieu( struct Point a,
                     struct Point b )
{
   a.x += b.x ;
   a.y += b.y ;
   a.x /= 2.0 ;
   a.y /= 2.0 ;
   return a ;
}
```

On travaille sur *a*, copie locale de l'argument d'entrée.

Passage de *struct* en valeur de retour.

L'appelant

```
struct Point a = {1.0,2.0}, b = {3.0,4.0};

struct Point m = milieu( a, b );
```

Opérations admises sur les *struct* (4)

Attention à l'affectation membre-à-membre

MaString.h

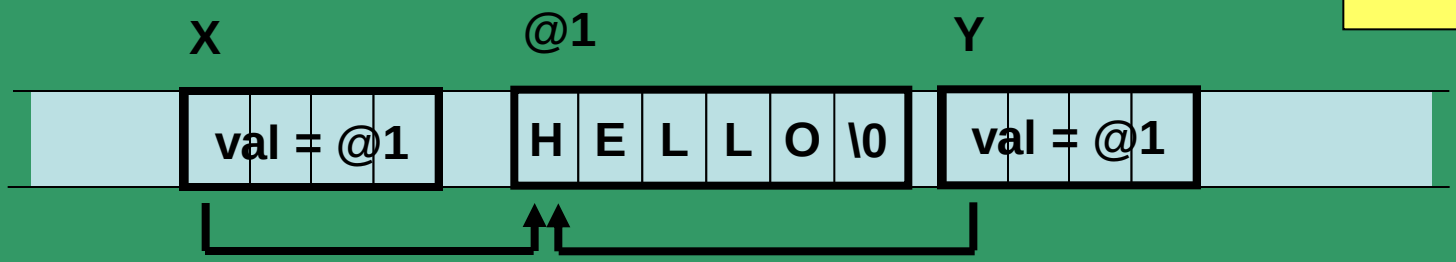
```
struct MaString {
   char  * val;
};
```

Main.cpp

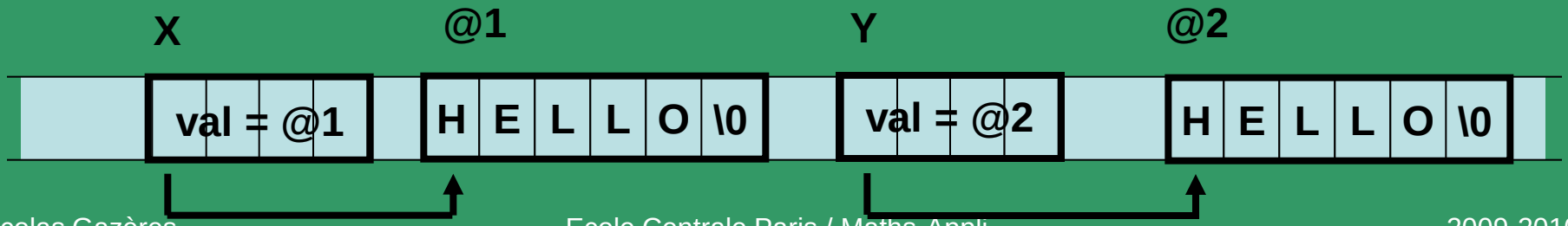
```
struct MaString X = { "HELLO" };
struct MaString Y = X ;
```

Le C++ gère cette affectation très différemment pour les classes !

- Configuration obtenue

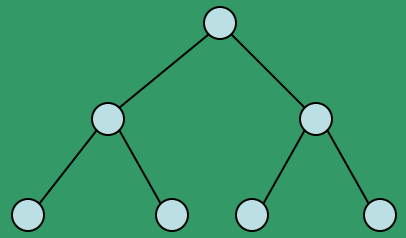


- Configuration généralement souhaitée



Opérations admises sur les *struct* (5)
Auto-référence via des pointeurs

- la définition d'une *struct X* peut contenir un membre de type « pointeur vers la *struct* X ».
- Exemple : la structure qui modélise le nœud d'un arbre binaire.



- Définition de la structure :

```
struct Node {
    struct Node * left;
    struct Node * right;
    …;
  } ;
```

Ne compile pas

```
struct Node {
    struct Node left;
    struct Node right;
    …;
  } ;
```

Opérations interdites sur les *struct*
La comparaison

- La comparaison d'instances de *struct* est interdite.
  - Le langage C ne définit pas d'opérateur == pour les *struct.*
    → ***erreur à la compilation***
  - le programmeur doit se définir une *fonction* spéciale pour la comparaison.

- L'addition, la multiplication, … de *struct* ne sont pas disponibles.

Tableaux d'instances de *struct*

- Exemple
  - le triangle

```
struct Triangle {
   struct Point sommets[ 3 ];
};
```

- Tableau initialisé

```
struct MaChaine { char * s; };

struct MaChaine mois[]
      = { { "Janvier" },
          { "Février" },
            …,
          { "Décembre" },
        };
```

On ne précise pas la taille du tableau (le compilateur calcule automatiquement)

Fin