Introduction au C++
Les classes

Pierre-Édouard Cailliau & Nicolas Gazères

Dassault Systèmes
Science & Corporate Research

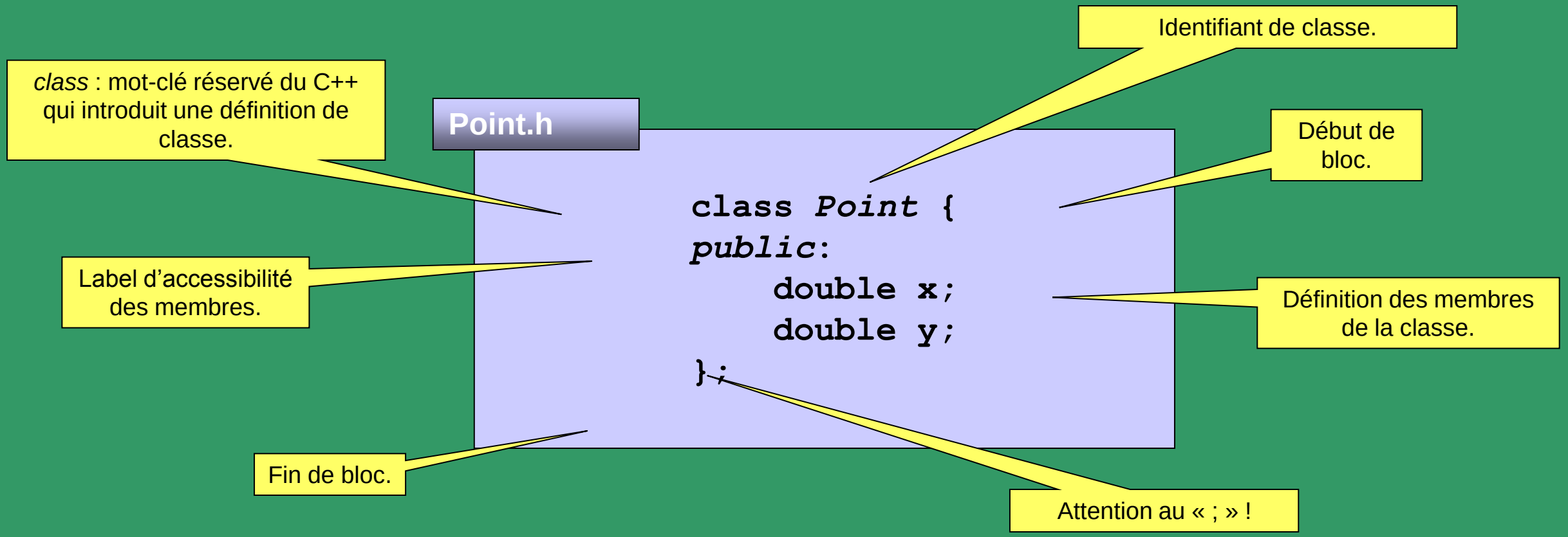
Abstraction

- Définition
  - Processus qui consiste à trouver le bon niveau de description pour un concept du domaine à modéliser informatiquement.
    - Les informations que l'on conserve ; celles que l'on décide d'ignorer.
      - Ex. mécanique du point *versus* géophysique.
- Les classes sont le mécanisme fourni par le C++ pour l'abstraction des concepts.
- Les classes permettent de construire des types utilisateurs.
  - Ils expriment dans la modélisation informatique les concepts du domaine d'étude.
  - Ils sont construits *par composition* :
    - à partir des types standards C++ ;
    - à partir d'autres classes.
- Une hiérarchie de types se construit ainsi :
  - Échelle d'abstraction croissante.

Définition de classe

Syntaxe

- La « *classe* » est le mécanisme offert par le langage C++ pour l'abstraction des données.
- On groupe dans une classe des données liées entre elles, plutôt que de les traiter indépendamment : *l'état interne*.



Déclaration d'une « instance de classe »

- Allocation sur la Pile
  - Syntaxe

mon_source.cpp

```
{
  Point P1;
  Point P2, P3;
  Point P[10];
  …
```

- On peut allouer autant d'instances de la classe que la mémoire le permet :
  - Toutes les instances partagent la même « structure » physique.
    - Analogie moule/pièce moulée.
  - Chaque instance possède son *état interne propre.*
    - *Les valeurs de ses données-membres.*
- À la différence des *struct* du C, il est n'est pas nécessaire en C++ de répéter le mot-clé *class* avec l'identifiant de celle-ci.

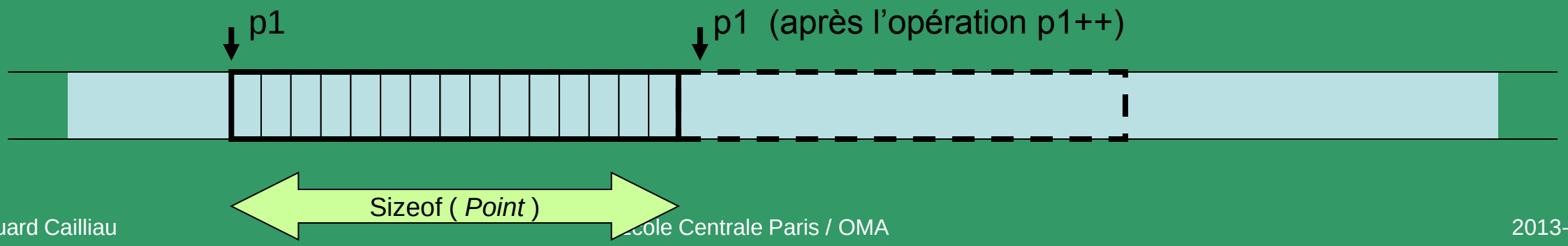
Manipulation par pointeur

- Chaque instance de classe possède une ***adresse mémoire.***
  - On peut la récupérer par l'opérateur &.
- L'adresse de l'instance de classe peut être manipulée via un pointeur de type « *pointeur sur Point* » :

mon_source.cpp

```
Point c1, *pc1 = &c1 ;
```

- L'opérateur ++ fonctionne sur les « pointeurs de classe ».
  - p1++ déplace le pointeur « après » l'instance de classe.
  - En octets, le décalage correspond à la taille de la classe telle qu'elle est donnée par l'opérateur *sizeof*.



Accès aux membres d'une classe (1/2)

- L'opérateur « point » (ou opérateur de sélection)
  - utilisable sur toute variable de type *classe*.
- Syntaxe : « variable . membre »
  - « membre » peut être une donnée ou une méthode
    - instance_de_classe.donnee ;
    - instance_de_classe.methode() ;
  - L'expression qui désigne le membre peut être utilisée partout où une expression du même type est attendue.

Point.h

```
class Point {
public:
    double x, y;
};
```

Déclaration par *l'auteur* de la classe.

main.cpp

```
main() {
  Point p1 ;
  … p1.x … ;
  … p1.y … ;
```

accès à la donnée.

Utilisation par un *client* de la classe.

Accès aux membres d'une classe (2/2)

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

```
Cercle *p1 = &c1 ;

… p1->r … ;
… p1->centre.x … ;
```

- On a l'équivalence suivante:

```
(*p1).r   ⇔   p1->r
```

conseil :
préférez celle-ci.

Les déclarations de méthodes (1/3)

- ***La première grande nouveauté par rapport au C***
  - On place également dans la définition d'une classe toutes les *opérations* autorisées sur les instances de la classe : les ***méthodes*** (ou fonctions membre).
  - La présence des méthodes sur une classe est un des mécanismes offerts par le C++ pour assurer **l'encapsulation** des données.
- Un programme objet « pur » n'utilise :
  - que des *instances de classes*…
  - …qui collaborent via des *appels de méthodes* sur ces instances.
- Il n'y a plus d'appels de fonctions extérieures à des classes (en théorie…).

- La classe est donc un tout cohérent :
  - son état interne (*données-membres*),
  - les opérations qu'elle offre à ses utilisateurs (*méthodes*)

Segment.h

```
class Segment {
public:
   double longueur() ;
private:
   Point a;
   Point b;
};
```

Nouveau label d'accessibilité des membres (voir plus loin…)

L'implémentation d'une méthode (2/3)

- Le plus souvent dans un fichier source (.cpp) séparé du fichier d'entête (.h).
- On préfixe le nom de la méthode par
  - le nom de la classe
  - le séparateur :: (dit « opérateur de *portée* »)
    - On dit que « la méthode est définie dans la portée de la classe ».
- Dans le corps de la méthode, on peut accéder aux données-membres sans les préfixer par le nom de classe (on est déjà dans la bonne portée).

Déclaration des méthodes de la classe dans le header.

Segment.h

```
class Segment {
public:
   double longueur() ;
private:
   Point a;
   Point b;
};
```

Implémentation des méthodes de la classe dans le fichier *source.*

Segment.cpp

```
double Segment::longueur()
{
   double dx = b.x-a.x ;
   double dy = b.y-a.y ;
   return sqrt(dx*dx+dy*dy);
}
```

L'invocation d'une méthode (3/3)

- Comme pour les données-membres, on utilise :
  - l'opérateur « point » (.) si l'instance est manipulée par valeur ;
  - l'opérateur « flèche » (->) si l'instance est manipulée par pointeur.

Point.h

```
class Point {
public:
  Point(double ix, double iy);
private:
  double x, y;
};
```

Segment.h

```
class Segment {
public:
   Segment( Point ia, Point ib );
   double longueur() ;
private:
   Point a;
   Point b;
};
```

main.cpp

```
main()
{
  Point a( 0, 1 );
  Point b( 2, 3 );

  Segment s(a,b);

  double len=s.longueur();
  …
```

Rq: en réalité, on passerait le Point par référence const…

Les opérateurs

- ***La deuxième grande nouveauté par rapport au C***
  - On peut également définir le sens d'un ***opérateur*** lorsqu'il est appliqué à des instances d'une classe.
  - Quasiment tous les opérateurs peuvent être redéfinis (+, *, ++, ==, !, ->, …).
  - Un opérateur de classe est une sorte de méthode ayant une syntaxe beaucoup plus naturelle.

LigneBrisee.h

```
class LigneBrisee {
public:
   void operator+=( Point p ) ;
   …
};
```

main.cpp

```
int main() {
  Point p1,p2,p3;
  LigneBrisee l(p1);
  l += p2;
  l += p3;
}
```

LigneBrisee.cpp

```
void LigneBrisee::operator+=( Point p ) {
  …;
  return;
}
```

Rq: en réalité, on passerait le Point par référence const…

Contrôle d'accès

- ***La troisième grande nouveauté par rapport au C***
  - Le *contrôle d'accès* est un des mécanismes offerts par le C++ pour assurer **l'encapsulation** des données**.**
- Les deux premiers spécificateurs d'accès sont:
  - ***public*:** le symbole peut être utilisé partout
  - ***private*:** le symbole ne peut être utilisé que par les méthodes de la classe.

MaClasse.h

```
class MaClasse {
public:
    double a;
    int m1( char *p );
private:
    int b;
    void m2( int i );
};
```

Interface publique.

Données privées

- Ils s'appliquent aux méthodes comme aux données-membres.

MaClasse.cpp

```
int MaClasse::m1( char *p ) {
   b++;          // OK
   m2( 4 );      // OK
   …;
}
```

main.cpp

```
MaClasse x1;
y=sin( x1.a );          // OK
x1.m1("xxx");           // OK
x1.b++ ;        // Compile KO !!
x1.m2(3);       // Compile KO !!
```

Encapsulation: on ne peut passer que par l'interface publique.

Contrôle d'accès

friend

- **Définition**
  - Le mot-clé *friend* permet d'étendre l'accessibilité des membres d'une classe à une (ou plusieurs) autre(s) classe(s).
  - Il s'utilise dans le fichier d'entête de la classe qui expose ses *membres privés*.
  - On peut déclarer *friend* toute une classe, ou bien simplement une méthode ou une fonction.

MaClasse.h

```
class MaClasse {

friend class MonAutreClasse;

private:
    int b;
    void m2( int i );
};
```

Déclaration *friend*

MonAutreClasse.cpp

```
int MonAutreClasse::m1( MaClasse mc )
{
   mc.b++;        // OK
   mc.m2( 4 );    // OK
   …;
}
```

L'implémentation de *MonAutreClasse* fait maintenant partie de l'interface publique.

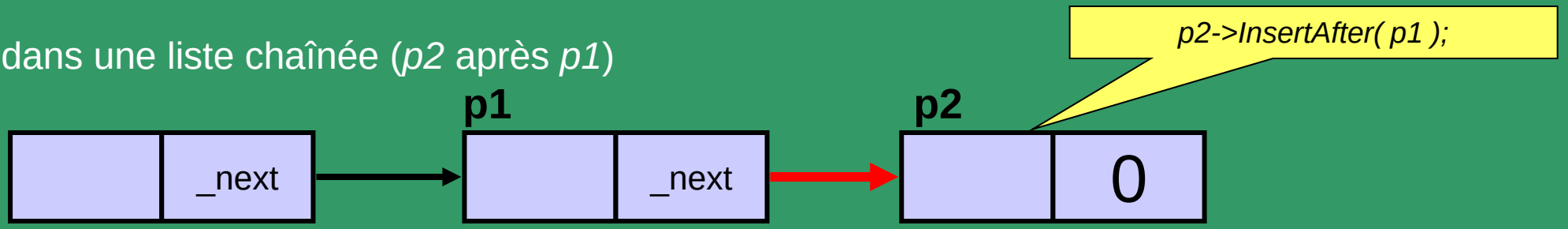
main.cpp

```
MaClasse mc;
mc.b++ ;          // Compile KO !!
mc.m2(3);         // Compile KO !!
```

Le main ne fait toujours pas partie de l'interface publique.

Objet courant de méthode

- Le besoin
  - L'implémentation d'une méthode a souvent besoin de savoir où se trouve en mémoire l'objet pour lequel on l'invoque...
- Exemple
  - Insertion dans une liste chaînée (*p2* après *p1*)



Maillon.h

```
class Maillon
{
public:
   Maillon();
   void InsertAfter( Maillon * );
   void SetNext( Maillon * );

private:
   Maillon * _next;
};
```

Maillon.cpp

```
void Maillon::SetNext( Maillon *iNext )
{
  _next = iNext;
}

void Maillon::InsertAfter( Maillon * p )
{
  p->SetNext( this );   // this=p2
  _next = 0;
}
```

La solution : le mot-clé *this*

- Définition
  - dans l'implémentation d'une méthode de classe, le pointeur *this* contient l'adresse de l'objet sur lequel la méthode a été invoquée.
- Le pointeur « *this* » est une variable locale implicite :
  - il n'a pas à être déclaré par le programmeur ;
  - il est de type « *pointeur constant sur l'objet de classe* » ;
  - il peut être utilisé comme n'importe quel autre pointeur.
- Il est indispensable pour l'opérateur d'affectation
  - Voir prochain cours.

MaClasse.cpp

```
void MaClasse::MaMethode()
{
   MaClasse * const this = … ;
   …
};
```

Pseudo-déclaration implicite de la variable *this* dans la méthode.

Main.cpp

```
main()
{
   MaClasse mc1;
   mc1.MaMethode();
}
```

Quand cet appel est exécuté, la variable *this* prend la valeur de l'adresse *&mc1*.

Les méthodes spéciales des classes

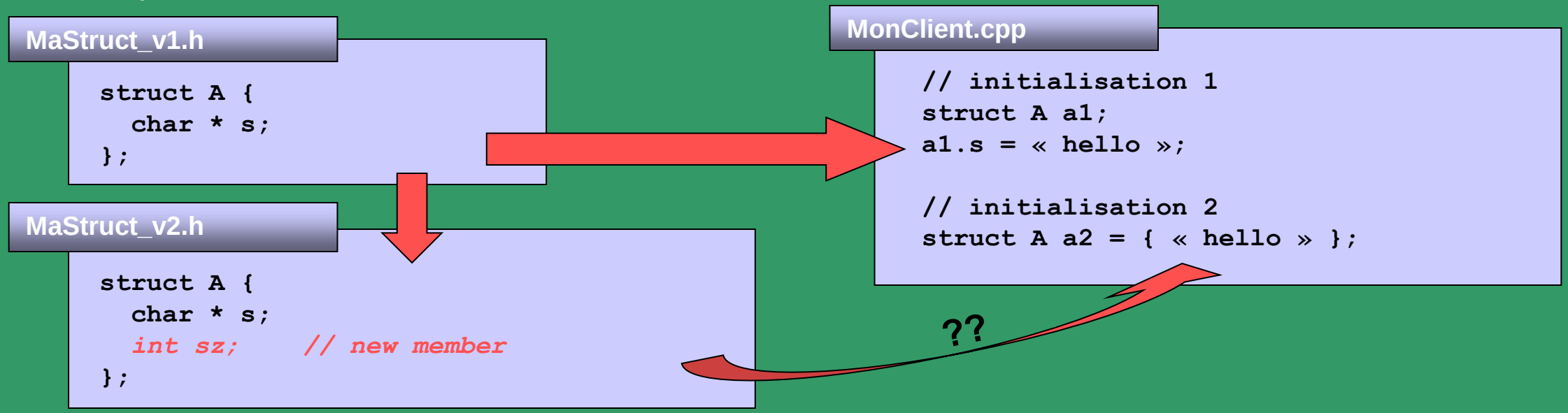
- En plus des méthodes définies par l'utilisateur sur la classe, il y a *quatre méthodes spéciales* :
  - **Le(s) constructeur(s)**
    - Responsables de l'initialisation correcte et complète des objets
    - *Les* constructeurs : il peut y en avoir plusieurs dans une classe.
  - ***Le Destructeur***
    - *cf. prochain cours.*
  - ***Le constructeur de copie***
    - *cf. prochain cours.*
  - ***L'opérateur d'affectation*** *(operator=)*
    - *cf. prochain cours.*

- Les méthodes spéciales ont une caractéristique :
  - Le compilateur génère du code par défaut pour ces méthodes si le programmeur ne le fait pas explicitement.

Constructeurs

Le problème de l'initialisation des structures en C

- Exemple



- Inconvénients de l'initialisation des structures C
  - La maintenance du code n'est pas sûre
    - En C, on peut fournir moins de valeurs qu'attendues.
    - En cas d'ajout d'une nouvelle donnée membre couplée aux précédentes ?
  - Le code d'initialisation des objets n'est pas centralisé.
    - L'étude d'impact d'un changement en C peut laisser passer des erreurs.
  - Le C ne gère nativement l'initialisation que par valeurs constantes.

Constructeur

- Principe
  - Renvoyer un objet (1) complètement créé et (2) intègre.
  - Centraliser le code d'initialisation des objets.
- Syntaxe
  - Un constructeur est une méthode qui porte le même nom que la classe.
  - La liste des paramètres est arbitraire, éventuellement vide.
  - Un constructeur n'a pas de valeur de retour.

Matrice2x2.h

```
class Matrice2x2 {
private:
   double a11, a12, a21, a22;
public:
   Matrice2x2() {
        a11 = a22 = 1.0;
        a12 = a21 = 0.0;
   };
};
```

La valeur initiale par défaut est la matrice Identité

main.cpp

```
int main()
{
  Matrice2x2 I;
  …
}
```

Constructeur

- Un constructeur
  - Permet d'initialiser simplement les objets
    - *ie*. en affectant des constantes aux données-membres.
  - Permet aussi des initialisations plus sophistiquées :
    - *calcul* arbitraire des données-membres
    - allocation mémoire sur le Tas, …
    - acquisition de ressources (fichiers, réseau, connections base, *locks*, …)
  - Facilite l'étude d'impact et l'évolutivité du code.
    - Si l'on ajoute une donnée-membre à la classe, il est suffisant de modifier les constructeurs pour que tout code client continue à bénéficier d'objets initialisés de manière complète et intègre.

- Un constructeur est appelé à chaque fois qu'un objet est créé :
  - pour les objets alloués sur la *Pile* ;
  - à l'appel des opérateurs *new* ou *new [ ]* *(voir prochain cours) ;*
  - lors des appels de fonction qui utilisent le *passage par valeur* ;
  - lors du *passage par valeur* de la valeur de retour d'une fonction.

Constructeur

- Une classe peut posséder *plusieurs* constructeurs.
  - Ils diffèrent par leur signature.
  - Le constructeur choisi par le compilateur pour une instanciation donnée dépend du nombre et des types de paramètres passés par l'utilisateur de la classe.
- Exemple

Matrice2x2.h

```
class Matrice2x2 {
public:
   // ctor 1 (= constructeur par défaut)
   Matrice2x2();

   // ctor 2
   Matrice2x2( double x );

   // ctor 3
   Matrice2x2( double a11, double a12,
               double a13, double a14 );
   …
};
```

main.cpp

```
main() {
  …
  // appelle ctor 1
  Matrice2x2 I ;

  // appelle ctor 2
  Matrice2x2 I( 3.0 );

  // appelle ctor 3
  Matrice2x2 I(1.0,2.0,3.0,4.0);
  …
```

Constructeur par défaut

- Définition
  - Le constructeur par défaut est un constructeur qui peut être appelé lorsque le code instancie un objet sans passer d'arguments.
- Règles
  - Si l'auteur d'une classe *A* définit un constructeur par défaut,
    alors le compilateur utilisera ce constructeur pour toute instanciation *sans arguments* d'un objet de type *A* (*cf.* **`Matrix2x2`**)
  - Si l'auteur de *A* ne définit pas de constructeur par défaut :
    - Si l'auteur de A ne définit strictement *aucun* constructeur :
      - alors c'est *le compilateur lui-même* qui fabrique un constructeur par défaut.
    - Si l'auteur de A définit au moins un autre constructeur
      (qui peut être par exemple le constructeur de copie, *cf. cours prochain*) :
      - le compilateur ne génère pas de constructeur par défaut ;
      - pour chaque instanciation d'un objet de classe *A*, le compilateur cherche à utiliser l'un de ces constructeurs « explicites » ;
      - si aucun constructeur « explicite » ne convient, il y a erreur de compilation.

Constructeur d'une classe ayant des objets-membres
(ie. des membres de type classe)

- Exemple

Roue.h

```
class Roue {
public:
   Roue( );
};
```

Cadre.h

```
class Cadre {
public:
   Cadre( );
};
```

Les composants

Le composé

Velo.h

```
class Velo {
public:
   Velo( … );
private:
   Roue  avant, arriere;
   Cadre cadre ;
};
```

Velo.cpp

```
Velo::Velo( … ) {
   // Construire les roues, la selle
   // Initialiser le vélo lui-même …
};
```

Constructeur d'une classe ayant des objets-membres

- *Règle C++ :*
  - *Les composants d'un objet doivent être complètement construits avant que le code du constructeur de l'objet soit exécuté.*

- Conséquence:
  - Quand il écrit le constructeur d'un objet ayant des objets membres, le programmeur a – *pour chaque membre* – deux possibilités:
    - 1/ Désigner explicitement le constructeur à utiliser
      - → *la liste d'initialisation*
    - 2/ Ne rien spécifier
      - Le compilateur insère alors un appel au constructeur par défaut.

Constructeur d'une classe ayant des objets-membres

Exemple 1: le programmeur désigne explicitement le constructeur à appeler

- Il existe une *notation spéciale* pour initialiser les membres.
  → *La liste d'initialisation*
  - En effet, les constructeurs des membres ne peuvent pas être appelés directement dans le corps du constructeur de la classe:
- Les constructeurs sont appelés dans l'ordre d'apparition des membres dans la *déclaration de classe* (pas dans la liste d'initialisation)

Velo.h

```
class Velo {
public:
   Velo( Couleur couleur,
         double diametre );
private:
   Roue  avant, arriere;
   Cadre cadre ;
};
```

Ordre d'apparition dans la déclaration de classe.

Velo.cpp

```
Velo::Velo( Couleur couleur,
            double diametre )
  : avant( diametre ),
    arriere( diametre ),
    cadre( couleur )
{
  … ;
}
```

Liste d'initialisation explicite.

Constructeur d'une classe ayant des objets-membres

Exemple 2: le programmeur ne spécifie rien

- Le compilateur insère alors un appel au constructeur par défaut.
  - Les constructeurs par défaut sont appelés dans leur ordre d'apparition dans la *déclaration de classe*.
  - Attention : Il peut y avoir une erreur à la compilation
    - si l'objet membre ne possède pas de constructeur par défaut,
    - ou si ce constructeur n'est pas accessible (ie. *private*).

Velo.h

```
class Velo {
public:
   Velo( … );
private:
   Roue  avant, arriere;
   Cadre cadre ;
};
```

Velo.cpp

```
Velo::Velo( … )
  : avant(),
    arriere(),
    cadre()
{
  … ;
}
```

liste d'initialisation *implicite*: appels aux constructeurs par défaut

Constructeur d'une classe ayant des objets-membres

Cas général: appel explicite pour certains membres et implicite pour les autres.

- Pour chaque donnée-membre pour laquelle l'appel au constructeur est *implicite*, le compilateur insère un appel au constructeur par défaut.
  - S'il existe et s'il est accessible
- Pour chaque donnée-membre pour laquelle le programmeur choisit *explicitement* un constructeur, le compilateur respecte le choix de constructeur fait par le programmeur.
  - Attention : cela ne fonctionne que si l'appel explicite au constructeur est fait *via la liste d'initialisation*.
  - Un appel explicite fait hors liste d'initialisation (ie. dans le corps du constructeur) vient en doublon de l'appel fait dans la liste d'initialisation.
    - Coût supplémentaire inutile !
- Dans tous les cas :
  - Les constructeurs sont appelés :
    - dans l'ordre de déclaration des membres *dans la déclaration de classe* (pas dans liste d'initialisation !)
    - avant le corps du constructeur de la classe

Constructeur d'une classe ayant des objets-membres

Cas général: appel explicite pour certains membres et implicite pour les autres

Velo.h

```
class Velo {
public:
   Velo( Couleur couleur, double diametre );
private:
   Roue  avant, arriere;
   Cadre cadre ;
};
```

Velo.cpp

```
Velo::Velo( Couleur couleur, double diametre )
   :   arriere( diametre )          avant( diametre ){
   … ;
}
```

Constructeur tel que codé par le programmeur

Constructeur final utilisé par le compilateur.

Velo.cpp

```
Velo::Velo( Couleur couleur, double diametre )
   :   avant(          )      ,  arriere(          )       ,    cadre( )
   … ;
}
```

Constructeur par défaut <u>généré</u>

Récapitulatif

- Si la classe a des membres qui sont des « classes avec constructeurs » :
  - Le constructeur par défaut généré par le compilateur est *public*
  - Ce constructeur initialise tous les membres de type « classe avec constructeur » :
    - dans leur ordre d'apparition dans la déclaration de classe
    - en appelant leur constructeur *par défaut.*
      - Si c'est possible…

- Si les seuls membres de la classe sont des données de type C classiques (*int, double, char*…*)
  - le constructeur par défaut généré est vide.
  - si on compile en mode optimisé, il peut ne même pas être généré.

Copy-Constructeur
Introduction rapide

- Le copy-constructeur d'une classe X
  - est un constructeur
  - permet de construire un objet de type X à partir d'un autre objet de type X
- Syntaxe

MaClasse.h

```
class MaClasse {
Public:
   MaClasse( const MaClasse & i );
   …
};
```

Pas de valeur de retour

Référence *const* en entrée

MonSource.cpp

```
// m2 et m3 sont des copies de m1
MaClasse m2( m1 );
MaClasse m3 = m1;
…
```

- Le copy-constructeur est utilisé:
  - soit explicitement (cf. exemple)
  - soit à chaque appel de fonction où on utilise le passage *par valeur*.
    - Pour un paramètre d'entrée
    - Pour la valeur de retour

Pointeur sur méthode

- Définition
  - Un « pointeur sur méthode » permet d'identifier et de manipuler une méthode d'une classe, indépendamment de son invocation.
  - On peut ainsi paramétrer un algorithme par une inconnue : une méthode de la classe, choisie et invoquée dynamiquement.
  - Le pointeur sur méthode conserve toute la puissance de typage de la signature de la méthode elle-même.
  - C'est une généralisation des « pointeurs sur fonction ».
    - Le pointeur sur méthode spécifie que l'opération à invoquer est associée à une certaine classe.
- Exemple

A.h

```
class A {
public:
   int m( double );
};
```

main.cpp

```
int main() {
  int (A::*pm)(double) = & A::m ;
  …
  A a;
  int res = a->*pm( 3.4 );
}
```

instance de pointeur sur méthode

Opérateur d'invocation

Opérateur d'adressage