Introduction au C++

Les méthodes spéciales des classes

Nicolas Gazères

Développeur Dassault Systèmes
ngs@3ds.com

Copie de *struct*

Rappel

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

Point.h

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

MonSource.cpp

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

- On a l'équivalence suivante:

```
struct Point P2 = P1;
```

Forme compacte (copie solidaire)

⇔

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

Forme développée équivalente

- *Problème*
  - Que se passe-t'il pour les membres de type pointeur ?...

Copie des *struct*

Attention à l'affectation membre-à-membre des pointeurs

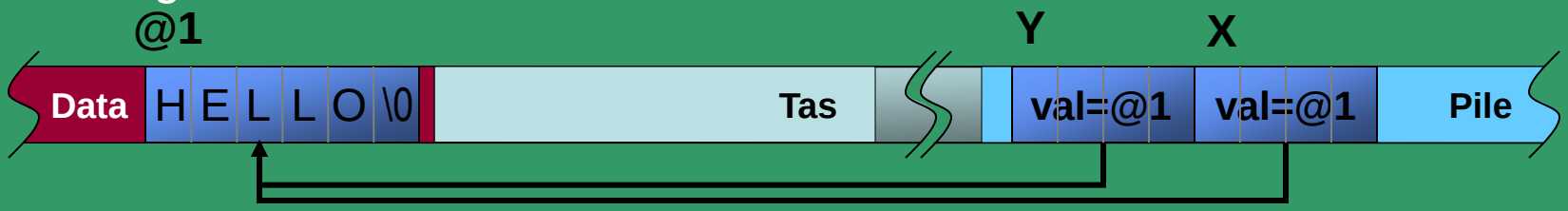
MaString.h

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

main.cpp

```
{
  struct MaString X = { « HELLO »};
  struct MaString Y = X ;
  …
```

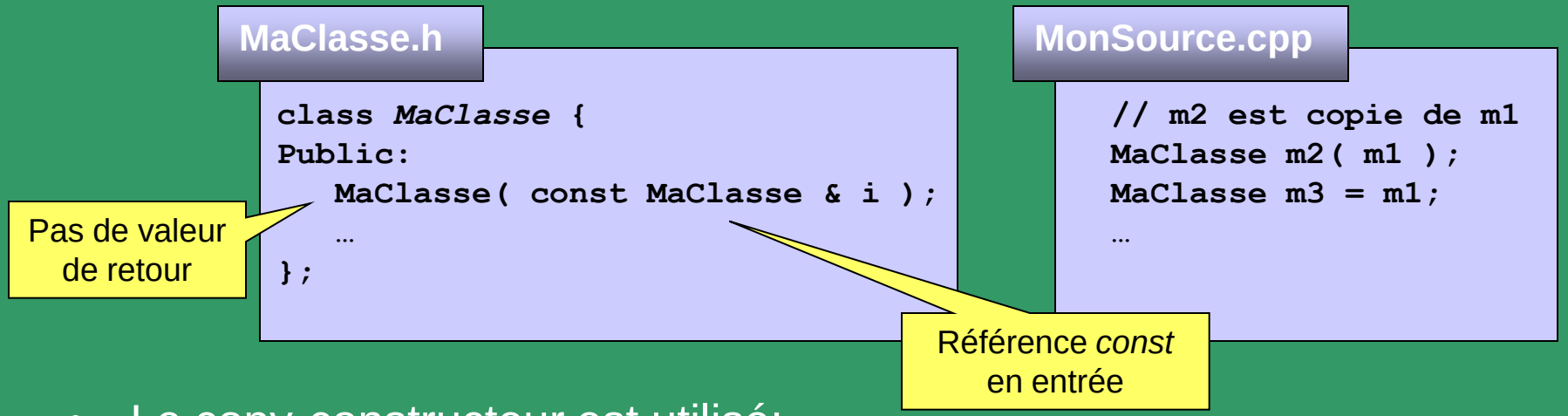
- Configuration obtenue



- Configuration généralement souhaitée:

Copy-Constructeur

- 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



- 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

Opérateur d'Affectation
operator=

- Le C++ offre un mécanisme au programmeur pour prendre le contrôle de l'affectation entre instances de classes :
  - *Il suffit de définir un opérateur d'affectation*.

- Syntaxe

MaClasse.h

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

Identifiant réservé

Renvoie une référence non-const

Référence *const* en entrée

- *Règle*
  - Le code de l'opérateur de copie sera exécuté lors de toute *affectation d'objet*.

MonSource.cpp

```
…
MaClasse mc1, mc2;
mc2=mc1 ;
…
```

Cette ligne invoque MaClasse::operator=

Méthodes Spéciales par Défaut

- **Copy-Constructeur**
  - Un constructeur de copie *par défaut* est généré par le compilateur si le programmeur n'en définit pas un lui-même.
    - Ce constructeur généré est *public*.
  - le comportement du constructeur de copie *par défaut* est :
    - D'appeler le constructeur de copie de tous les membres qui en ont un
    - De copier bit-à-bit sinon.
- **Opérateur d'Affectation (*operator*=)**
  - Un opérateur d'affectation *par défaut* est généré par le compilateur si le programmeur n'en définit pas un lui-même.
    - Cet opérateur généré est *public*.
  - le comportement de l'opérateur d'affectation *par défaut* est:
    - D'appeler l'opérateur d'affectation de tous les membres.
    - De copier bit-à-bit sinon.
- **Conclusion**
  - Comportement *par défaut* = homogène à celui des *struct* C
    → Conforme au principe de « plongement » du C dans le C++

Introduction au C++

Le problème dit « des gros objets »

Le problème des « gros objets »
Exemple

- Une classe *String* qui contient:
  - un pointeur sur caractères « char * »
  - la longueur de la chaîne « int »
- Le constructeur doit effectuer sa propre ***copie privée*** de la chaîne.

MaString.h

```
class MaString {
public:
   MaString( char * iStr );

private:
   char *       val;
   unsigned int sz;
};
```

MaString.cpp

```
MaString::MaString( char * iStr )
{
  sz = strlen( iStr );
  val = new char [ sz+1 ];
  strcpy( val, iStr );
  val[sz]='\0';
}
```

Le problème des « objets »

Utilité du cons

- Exemple

MaString.cpp

```
MaString::MaString( char * iStr )
{
  sz = strlen( iStr );
  val = new char [ sz+1 ];
  strcpy( val, iStr );
  val[sz]='\0';
}
```

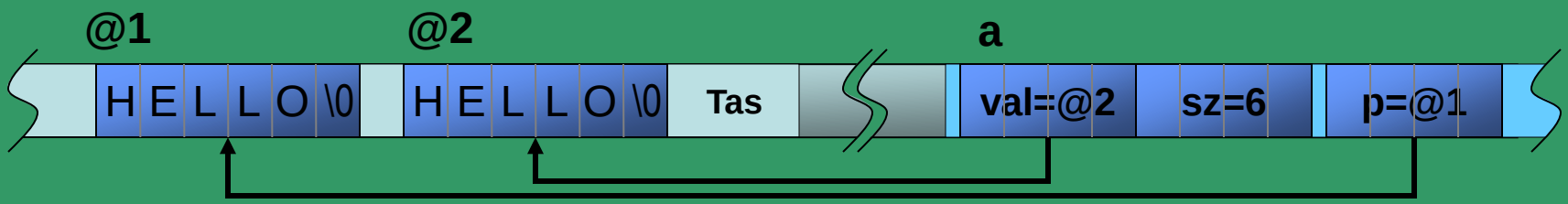
main.cpp

```
{
  const char *p = new
  strcpy( p, "hello" );

  MaString a( p );
  delete [] p;
```

désallocation toujours possible après construction de *a*.

- Vue mémoire



Le problème des « gros objets »

En l'absence de destructeur défini par le programmeur

main.cpp

```
{
    MaString a( "hello" );
    …;
}
```

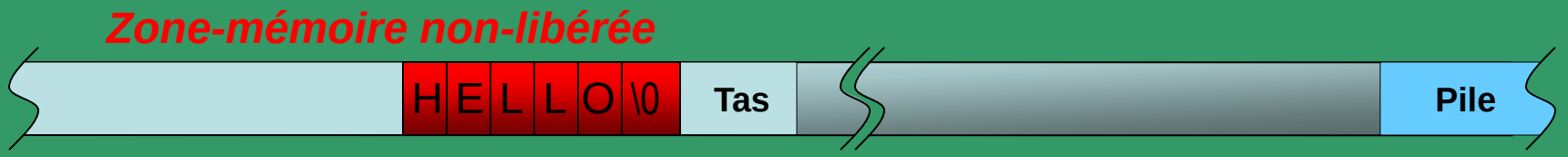
Passage par le destructeur *par défaut.*

pas de libération de la mémoire

Vue mémoire

- *a* est une variable locale allouée sur la *Pile*.
- Le constructeur effectue sa propre copie privée de « HELLO » en mémoire libre (Tas).

Le problème des « gros objets »
Utilité du destructeur



- *En l'absence de destructeur défini par le programmeur*
  - le destructeur par défaut n'appelle pas *delete [ ]*.
    - Alors que le constructeur a fait un *new [ ]*
  - La zone mémoire dans le Tas n'est pas libérée.
  - Comme le symbole *a* n'est disponible que dans le bloc dont on vient de sortir, l'oubli est *irréversible*.
    → **fuite-mémoire**.

- Les fuites-mémoires sont à éviter absolument.
  - si on les laisse s'accumuler, le programme s'exécute de plus en plus lentement et finit par planter.
    → un programme doit gérer au plus juste ses ressources-mémoire.

Le problème des « gros objets »
Utilité du destructeur

- Solution
  - un destructeur qui appelle *delete []*

MaString.h

```
class MaString {
public:
   MaString( char * str );
   ~MaString();

private:
   char *       _str;
   unsigned int _sz;
};
```

MaString.cpp

```
MaString::~MaString()
{
  delete [] _str;
}
```

Le problème des « gros objets »
Utilité du destructeur

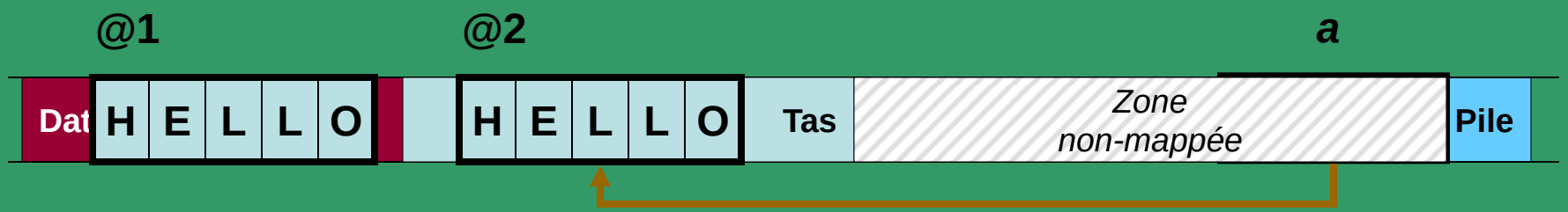
- Retour à l'exemple
  - Avec un destructeur qui appelle delete []

```
{
    MaString a( "hello" );
    …;
}
```

copie → allocation mémoire

Bonne libération de la mémoire

- Vue mémoire
  - A présent, la copie privée de « Hello » sur le Tas est désallouée par le code du destructeur de *a* en sortie de bloc.



Le problème des « gros objets ».

Avec un constructeur de copie par défaut

- Illustration: copie membre-à-membre

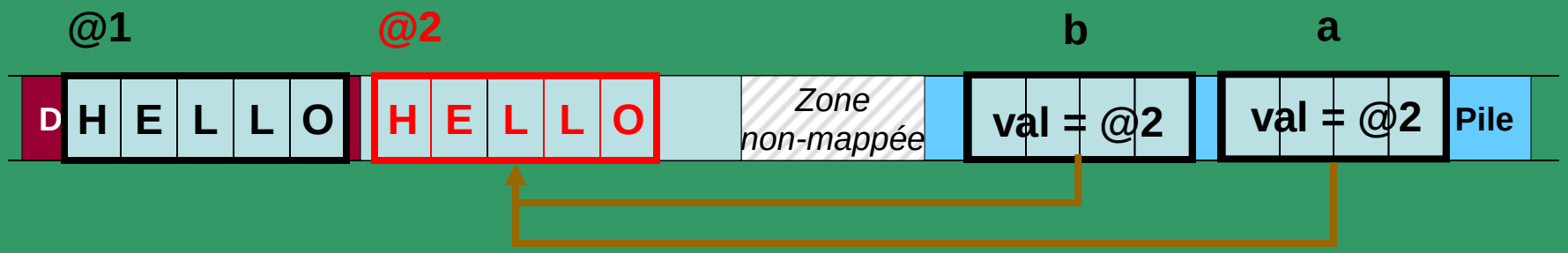
```
{
    MaString a( "hello" );
    MaString b( a );
    …;
}
```

copie → allocation mémoire

copie de pointeurs

Sortie de scope: double appel du destructeur

- Vue mémoire



Le problème des « gros objets »
Inconvénients du constructeur de copie par défaut

- Sans constructeur de copie
  - Pour chaque objet (*a* et *b*), le destructeur libère la mémoire de sa copie privée en sortie de bloc.
  - La *même* zone-mémoire, pointée par deux objets, est *désallouée deux fois* !
    → corruption du Tas.
    → risque d'erreur à l'exécution.

Le problème des « gros objets »
Définition explicite du constructeur de copie

- Solution
  - Programmer un constructeur de copie qui *duplique* la zone mémoire sous-jacente dans le Tas.

MaString.h

```
class MaString {
public:
   MaString( const MaString & other );
   …
};
```

MaString.cpp

```
MaString::MaString( const MaString & other ) {
  sz = other.sz;
  val = new char [ sz+1 ];

  // copie « sécurisée »
  strncpy( val, sz, other.val );
}
```

Le problème des « gros objets »
Utilité du constructeur de copie

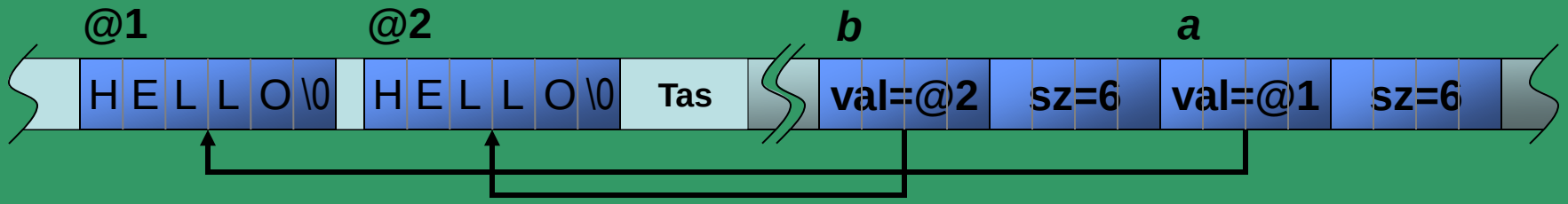
- Retour à l'exemple
  - avec un constructeur de copie défini par le programmeur, cette fois

```
{
    MaString a( "hello" );
    MaString b( a );
    …;
}
```

Changement !
duplication de la zone mémoire pointée

- Vue mémoire
  - Les destructeurs sont appelés dans l'ordre inverse des constructeurs

Le constructeur de *a* effectue sa propre copie privée de "hello" en mémoire libre (tas).



Le destructeur de *b* désalloue la copie privée de *b*

Le destructeur de *a* désalloue la copie privée de *a*

***duplique* la zone mémoire pointée.**

Le problème des « gros objets »

Avec un opérateur d'affectation par défaut

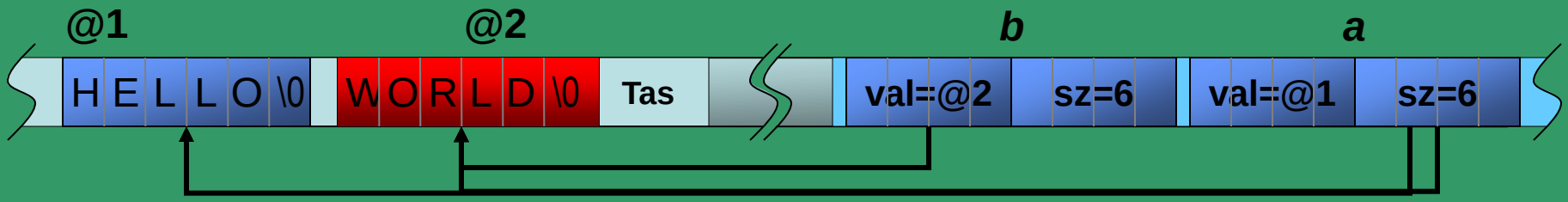
- Exemple

```
{
    MaString a( "hello" );
    MaString b( "world" );
    …;
    a = b;
}
```

copie → allocation mémoire

Sortie de scope: double appel du destructeur

- Vue mémoire



Le destructeur de *b* libère sa copie privée.→ *double désallocation* !

Le problème des « gros objets »

Inconvénients de l'opérateur d'affectation par défaut

- Sans opérateur d'affectation défini par l'utilisateur
  - La chaîne dans l'objet *b* pointe vers la même zone-mémoire que la chaîne de l'objet *a*.
  - Le destructeur libère deux fois la mémoire en sortie de bloc.
    → corruption du tas
  - La copie de pointeur perd l'ancienne valeur pour b
    → fuite-mémoire

Le problème des « gros objets »
Utilité de l'opérateur d'affectation

- Solution : un opérateur d'affectation défini explicitement:
  - Qui désalloue la zone mémoire précédemment occupée par le destinataire.
  - Qui duplique la zone mémoire associée à la nouvelle valeur.

MaString.h

```
class MaString {
public:
   MaString & operator=( const MaString & other );
   …
};
```

MaString.cpp

```
MaString & MaString::operator=( const MaString & other )
{
   delete [] val;
   sz = other.sz;
   val = new char [ sz+1 ];
   strncpy( val, sz, other.val );
   return *this;
}
```

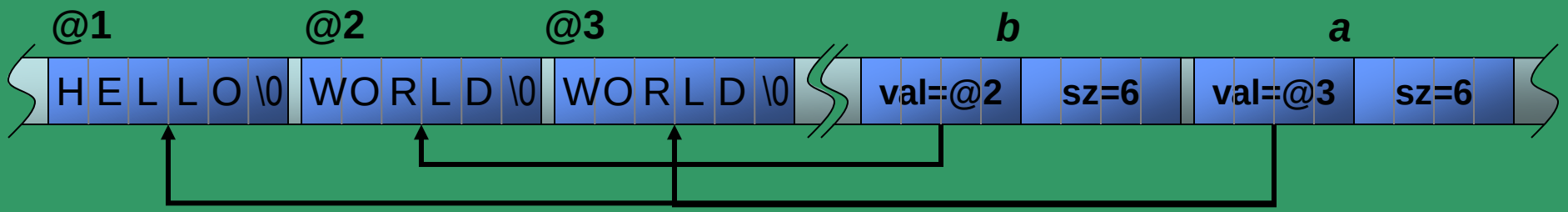
Utilité de *this*

Le problème des « gros objets »

Avec un opérateur d'affectation défini par le programmeur

```
{
   MaString a( "hello" );
   MaString b( "world" );
     ;
   a = b;
}
```

```
MaString &
 MaString::operator=(const MaString & other )
{
   delete [] val;
   sz = other.sz;
   val = new char [ sz+1 ];
   strncpy( val, sz, other.val );
   return *this;
}
```



Appel du destructeur d'affectation de b dans a → libération de sa copie privée de « WORLD »

Opérateur d'affectation

Attention piège !

- Protéger l'affectation contre **`a=a;`**
- Sans protection
  - Le *delete[]* détruit la zone-mémoire pointée par la cible, qui est également celle à copier !
    → au moment de la copie, la zone-mémoire n'existe plus !
- Version correcte
  - Effectuer le traitement *sauf* si la cible est identique à la source

MaString.cpp

```
MaString & MaString::operator=( const MaString & other )
{
   if ( this != &other ) {
      delete [] _str;
      _sz = other._sz;
      _str = new char [ _sz+1 ];
      strncpy( _str, _sz, other._str );
   }
   return *this;
}
```

Se protéger contre *a=a;*

Le problème des « gros objets »
Conclusion

- Les quatre méthodes spéciales sont:
  - **Le(s) Constructeur(s)**:
    Création d'un objet autonome à partir d'une zone-mémoire fournie par l'appelant (par pointeur ou par des spécifications de taille):
    - Soit le constructeur prend possession de la zone-mémoire,
    - Soit il s'en crée sa propre copie locale.
  - **Le Destructeur**:
    Libération des zones de mémoire sous-jacentes
    - éviter les fuites-mémoire.
  - **Le Copy-Constructeur**:
    Duplication des zones de mémoire sous-jacentes
    - éviter que deux objets ne partagent la même zone-mémoire (ce qui a toutes les chances d'aboutir à une corruption du tas).
  - **L'Opérateur d'affectation**:
    Remplacement des zones de mémoire sous-jacentes
    - éviter que deux objets ne pointent la même zone-mémoire (ce qui a toutes les chances d'aboutir à une corruption du tas).
    - éviter les fuites-mémoire.

Le problème des « gros objets »
Recommandations

- Quand le programmeur doit-il redéfinir les quatre méthodes spéciales des classes ?
  - Quand il y a un pointeur-membre
  - Vers un bloc de mémoire situé dans le tas
    - De taille variable
  - Et que le cycle de vie de ce bloc est lié au cycle de vie de l'objet qui le pointe.