POLYTECH NICE-SOPHIA

UNIVERSITÉ CÔTE D'AZUR Inria i3S

A <Basic> C++ Course

3 – Allocation dynanique et constructeurs

Julien Deantoni

KAIROS

Plan

- Allocation dynamique de mémoire
- Les constructeurs / destructeurs
  - Constructeurs
    - Par défaut (synthétisé et explicite)
    - Transtypage
    - Parametres
    - Parametres par defaut
    - De copie (synthétisé et explicite)
  - Destructeurs

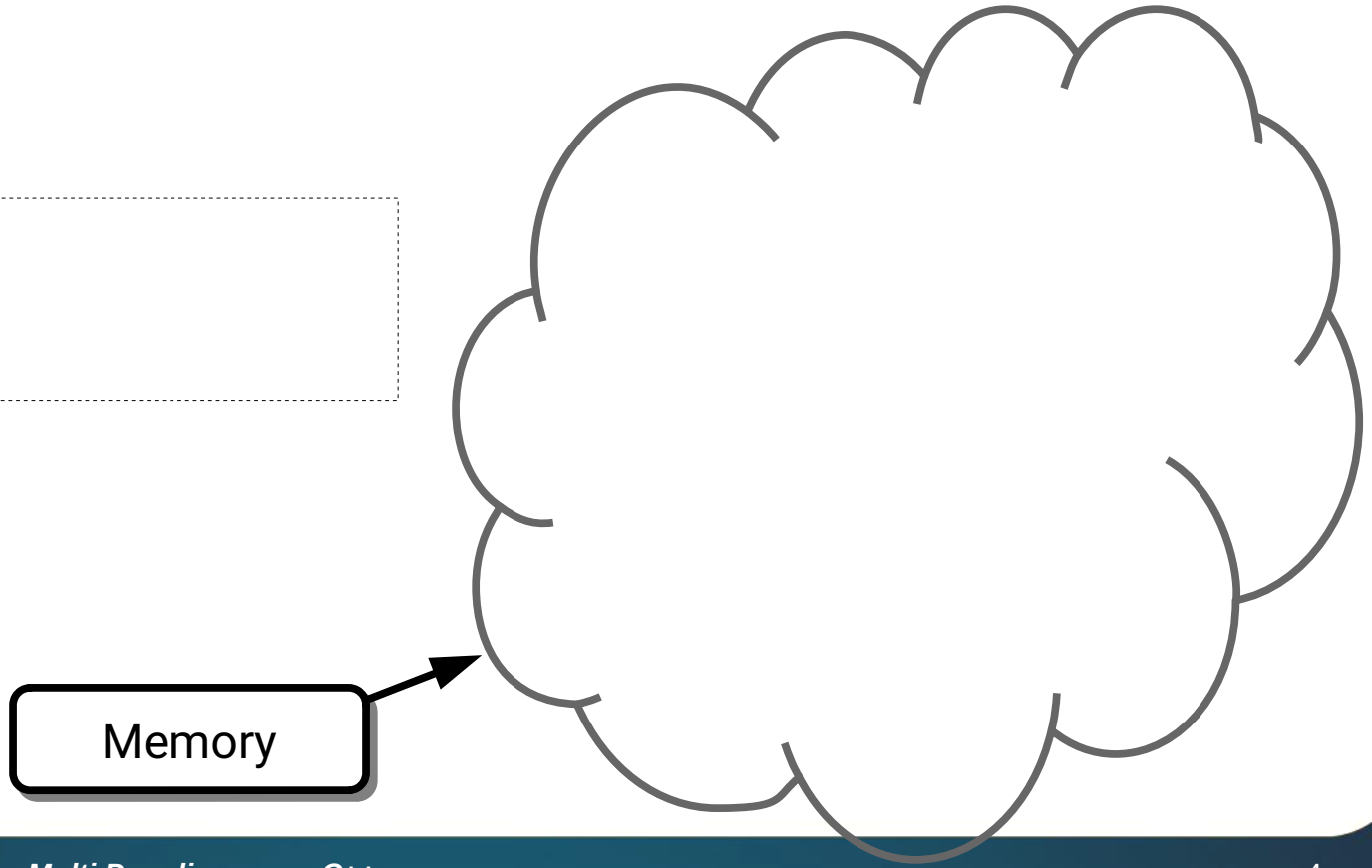
Memory concerns...

- Il existe deux manière de créer des objets en mémoire : statique et **dynamique**.
- Nous avons détaillé la création statique d'objet.
- Maintenant, passons à la réservation dynamique de mémoire.
- Les représentations qui suivent ne sont pas contractuelles mais abstraient la manière dont la mémoire est gérée.

Memory concerns...

- Les variables / objets sont stockés en mémoire

```
main(){

}
```

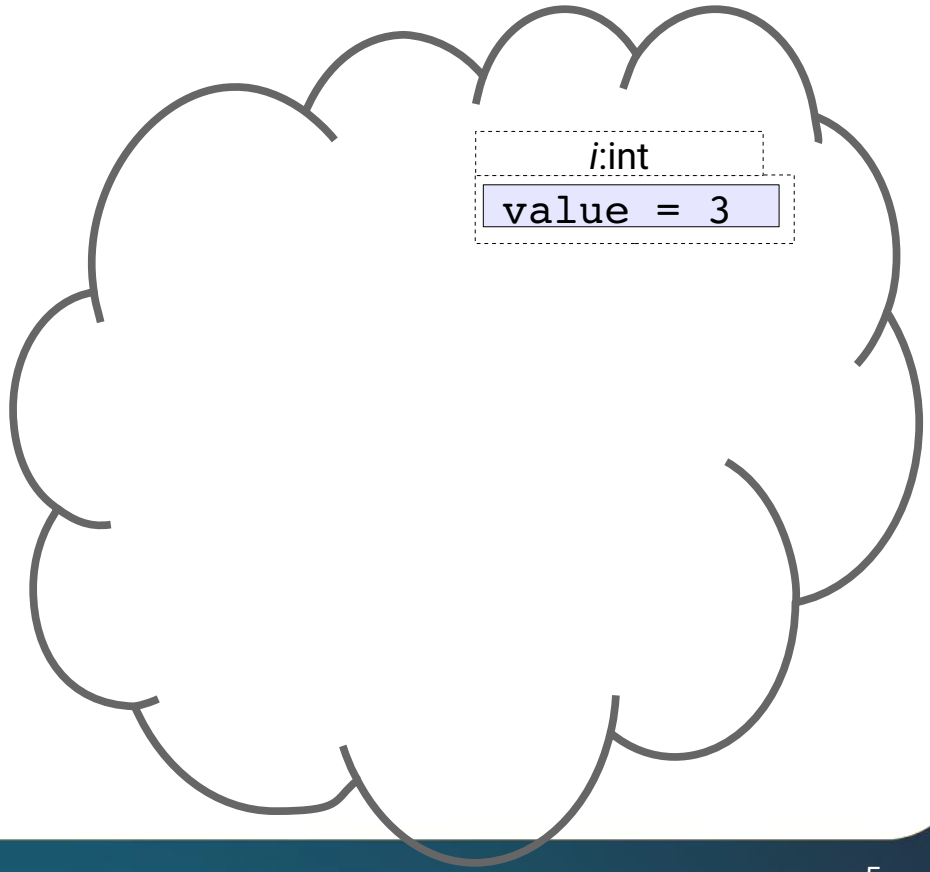


Memory concerns...

en statique...

- `anInt` est local à la fonction et détruit à la fin

```
int incremente(int anInt){
  anInt = anInt + 1;
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
}
```

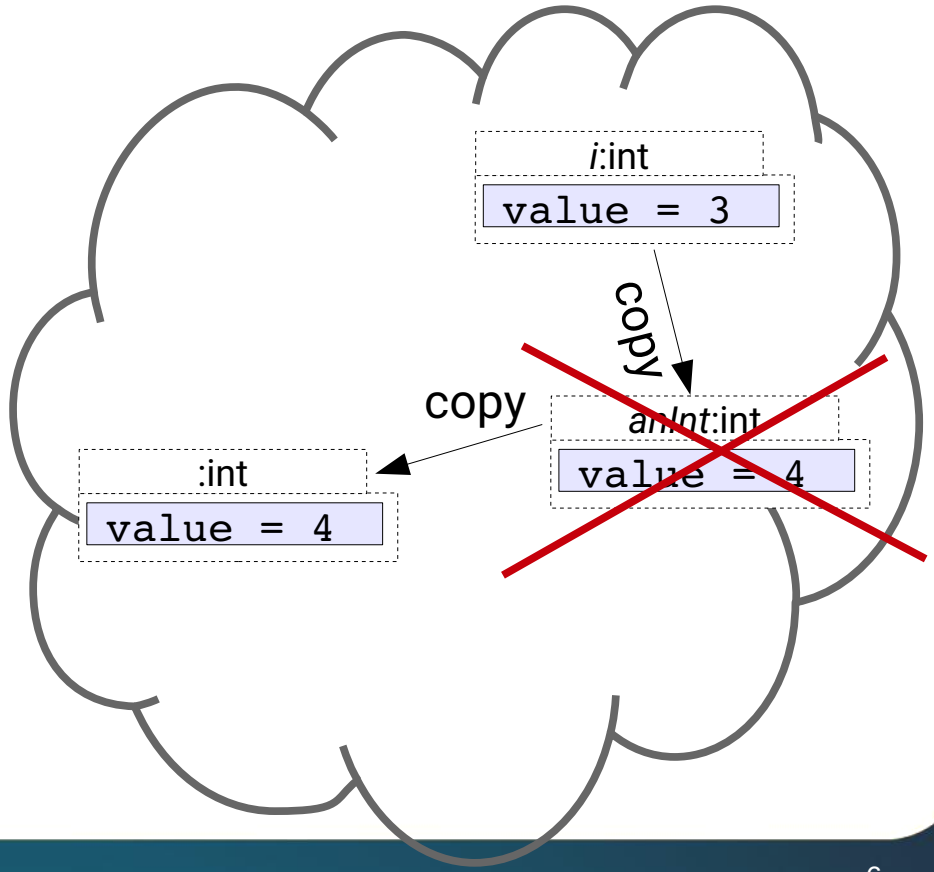


Memory concerns...

en statique...

- `anInt` est local à la fonction et détruit à la fin

```
int incremente(int anInt){
  anInt = anInt + 1;
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
}
```



Les variables/objets dont la mémoire est allouée statiquement sont détruits à la fin du bloc de déclaration

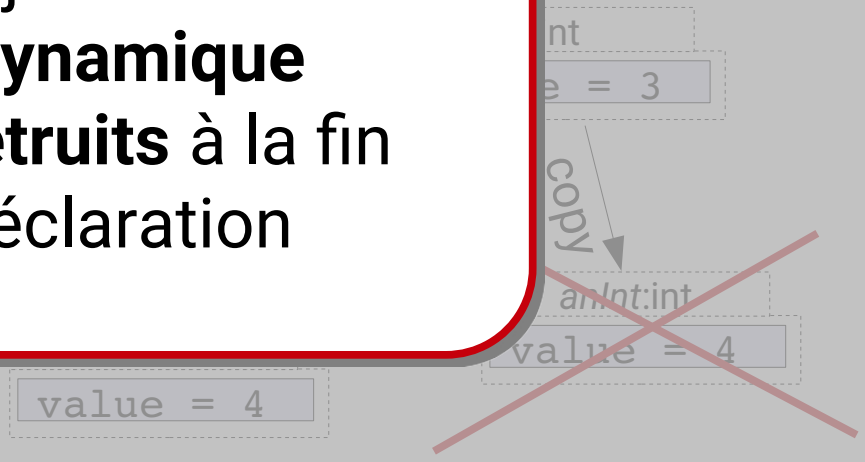
Memory concerns...

en dynamique...

- `anInt` est local à la fonction et détruit à la fin

```
int incremente(
  anInt = anInt
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
}
```

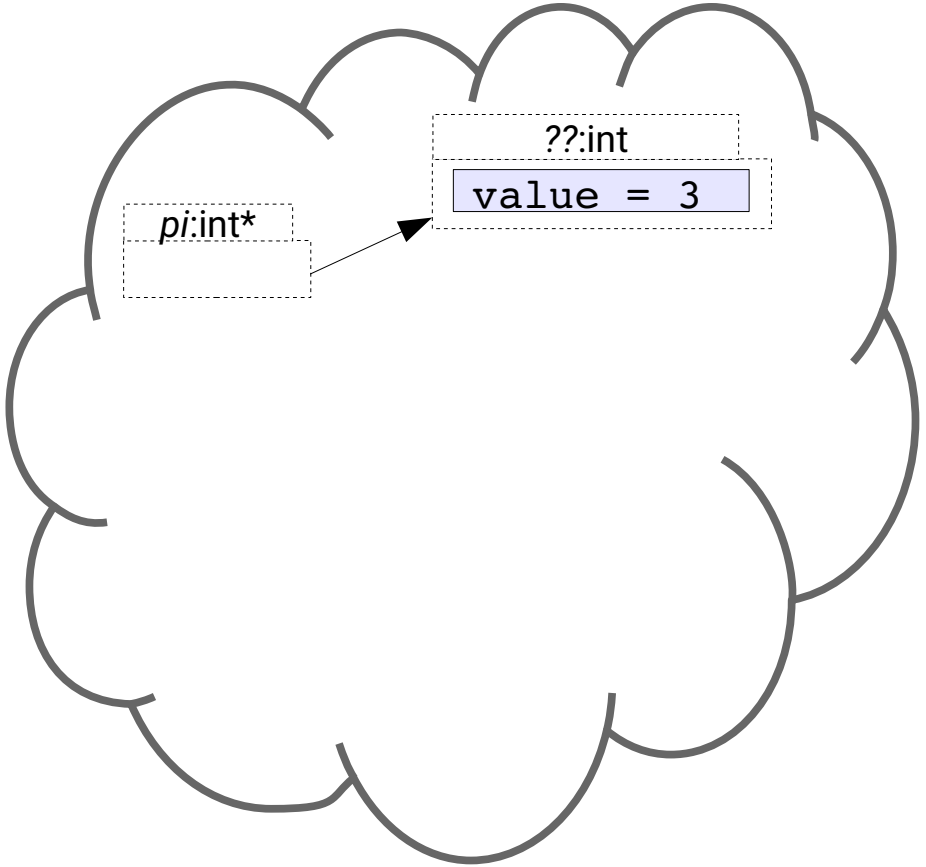
Les variables/objets reservés de manière **dynamique** **ne sont PAS détruits** à la fin du bloc de déclaration



Memory concerns...

- Reservation de mémoire :
  - Operateur **new**

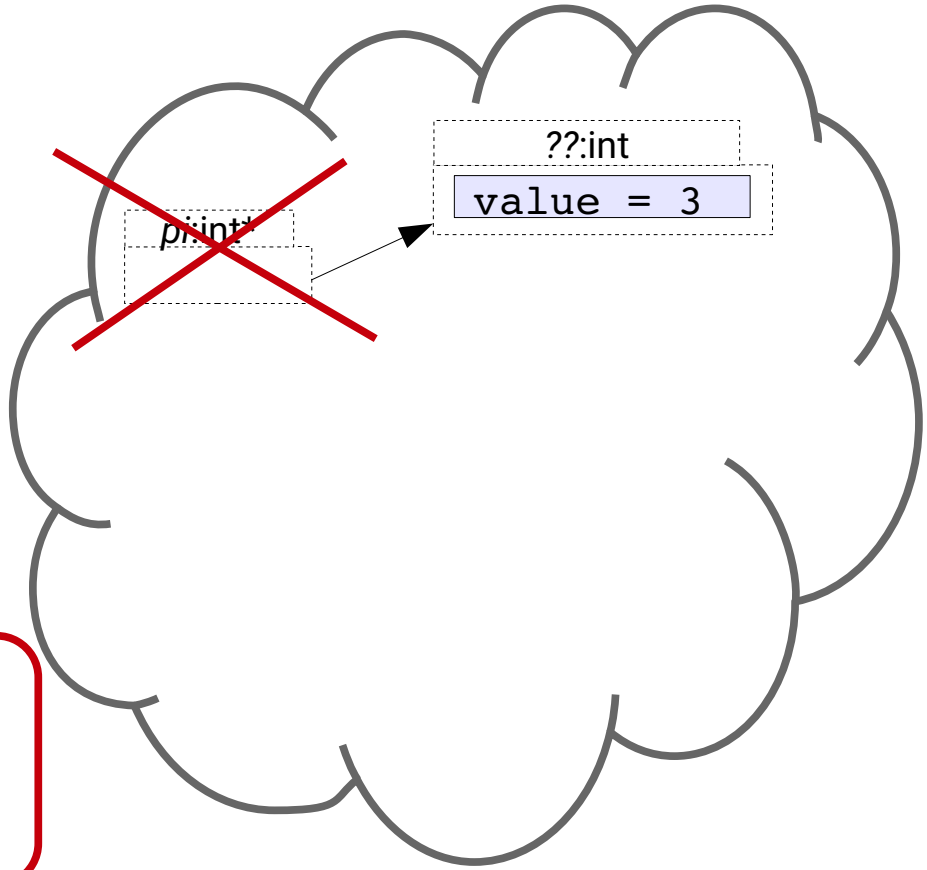
```
main(){
 int* pi = new int(3);
}
```



Memory concerns...

- Reservation de mémoire :
  - Operateur **new**

```
main(){
 int* pi = new int(3);
}
```

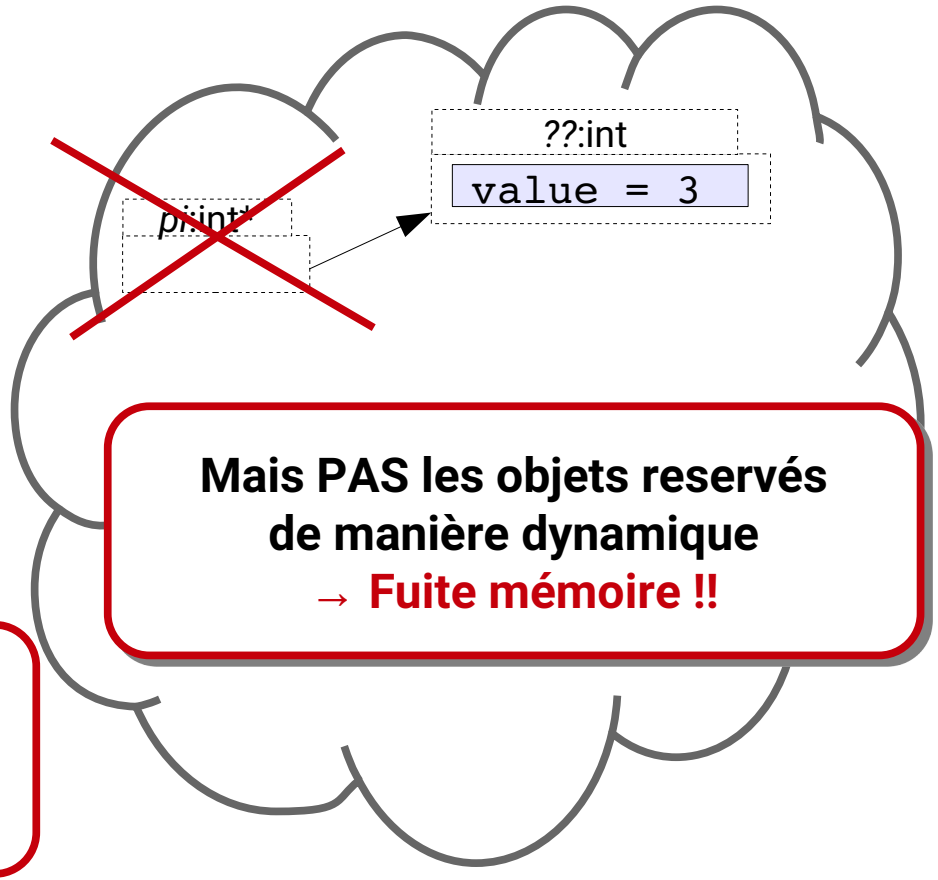


Les variables/objets dont la mémoire est allouée statiquement sont détruits à la fin du bloc de déclaration

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**

```
main(){
 int* pi = new int(3);
}
```



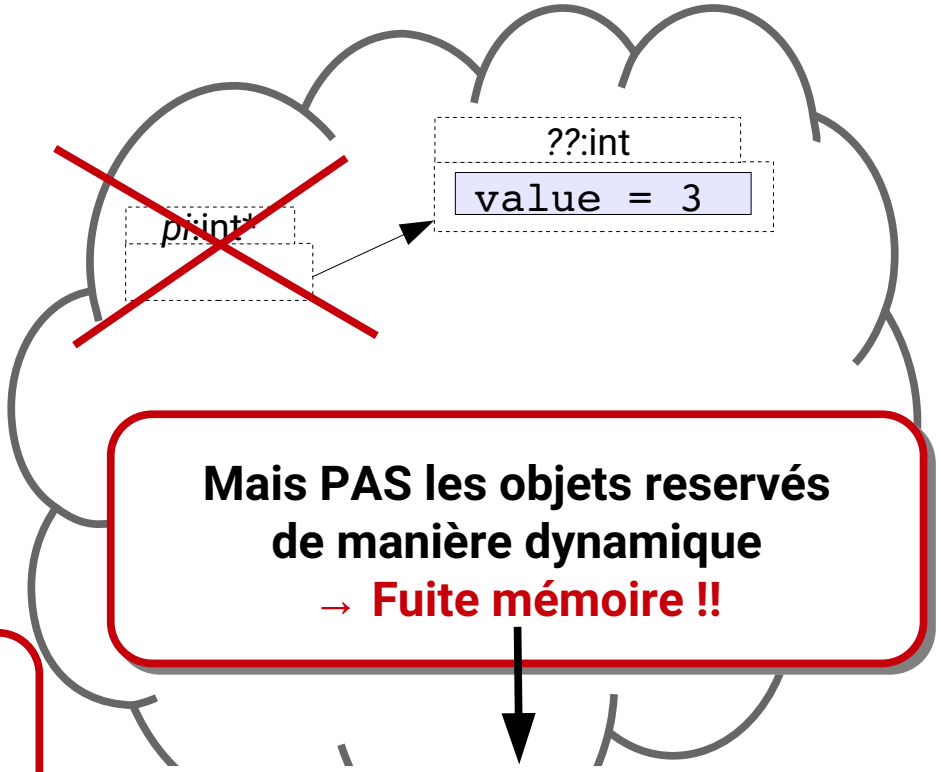
Mais PAS les objets reservés de manière dynamique
→ Fuite mémoire !!

Les variables/objets dont la mémoire est allouée statiquement sont détruits à la fin du bloc de déclaration

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**

```
main(){
 int* pi = new int(3);
}
```



Les variables/objets dont la mémoire est allouée statiquement sont détruits à la fin du bloc de déclaration

Les objets restent en mémoire
(jusqu'au prochain redémarrage?)

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**

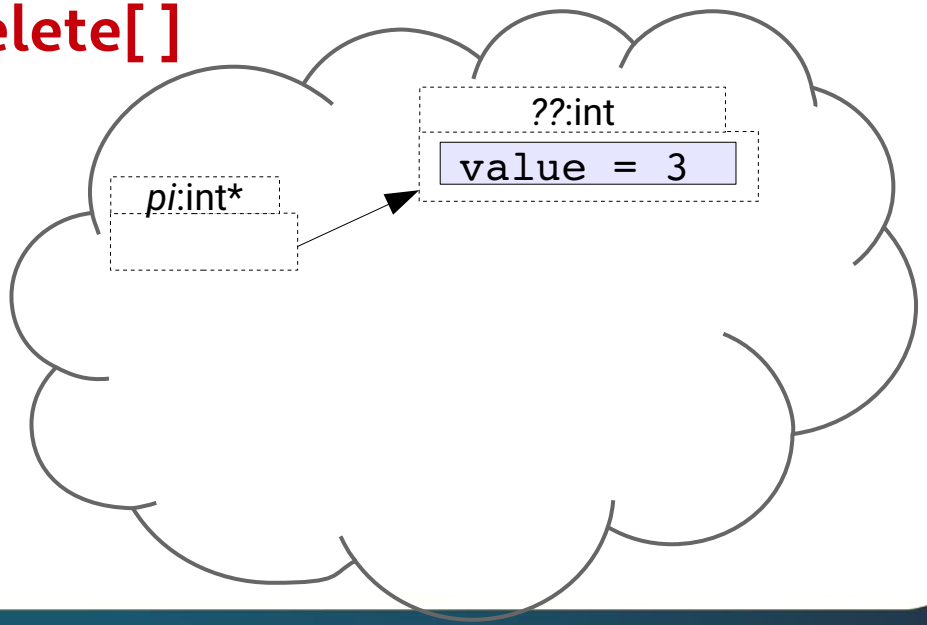
Il faut détruire explicitement
la mémoire allouée explicitement

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
}
```

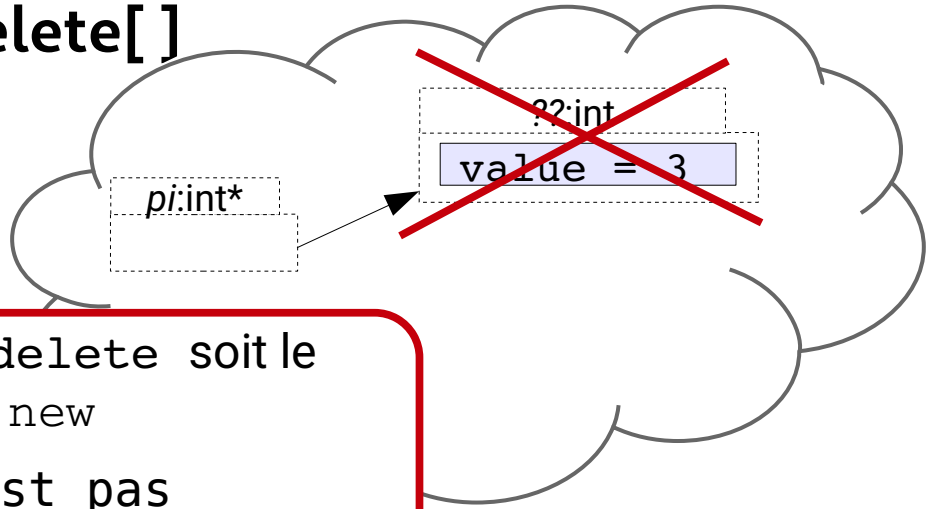


Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
   delete pi;
}
```



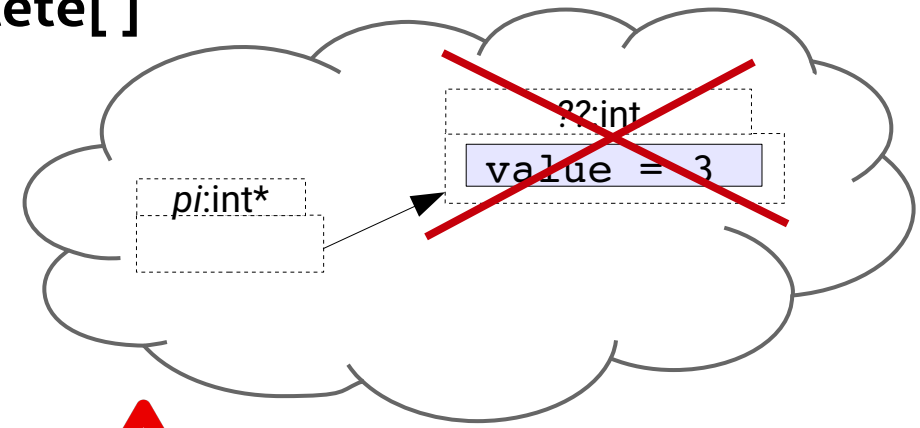
- Il faut que l'opérande de `delete` soit le résultat d'un (précédent) `new`
- `delete nullptr` n'est pas dangereux

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
   delete pi;
}
```



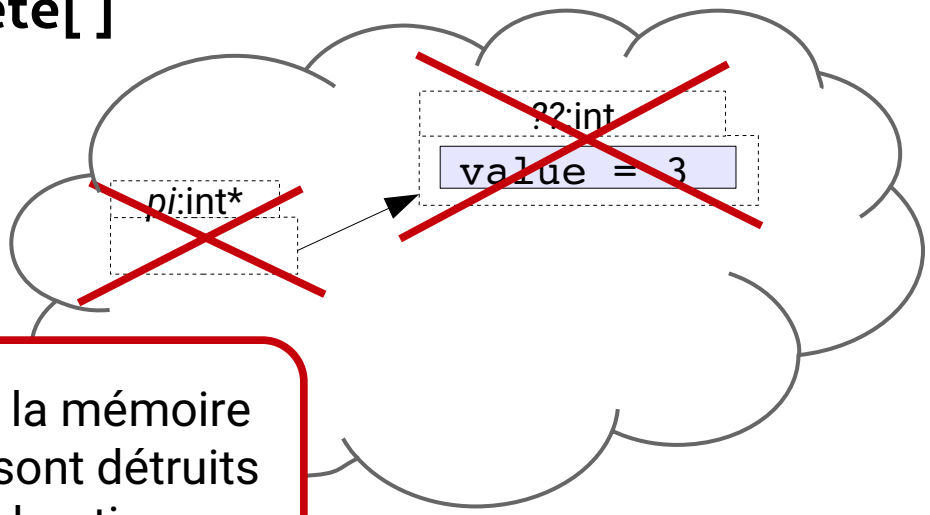
Attention `pi` peut exister sans
l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
   delete pi;
}
```



Les variables/objets dont la mémoire
est allouée statiquement sont détruits
à la fin du bloc de déclaration

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
   delete pi;
   cout << (*pi) << endl;
}
```

```
jdeanton@FARCI:$./executable
A votre avis ??????
jdeanton@FARCI:$
```

Attention `pi` peut exister sans
l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
   delete pi;
   cout << (*pi) << endl;
}
```

```
jdeanton@FARCI:$./executable
0
jdeanton@FARCI:$
```

Attention `pi` peut exister sans l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
   delete pi;
   cout << (*(pi+4)) << endl;
}
```

```
jdeanton@FARCI:$./executable
????
jdeanton@FARCI:$
```

Attention `pi` peut exister sans
l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
   delete pi;
   cout << (*(pi+4)) << endl;
}
```

```
jdeanton@FARCI:$./executable
0
jdeanton@FARCI:$
```

Attention `pi` peut exister sans
l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   string* pi = new string("toto");
   delete pi;
   cout << (*(pi)) << endl;
}
```

```
jdeanton@FARCI:$./executable
????
jdeanton@FARCI:$
```

Attention `pi` peut exister sans
l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   string* pi = new string("toto");
   delete pi;
   cout << (*(pi)) << endl;
}
```

```
jdeanton@FARCI:$./executable
Segmentation fault (core dumped)
jdeanton@FARCI:$
```

Attention `pi` peut exister sans l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   string* pi = new string("toto");
   delete pi;
   cout << (*(pi+4)) << endl;
}
```

```
jdeanton@FARCI:$./executable
????
jdeanton@FARCI:$
```

Attention `pi` peut exister sans
l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   string* pi = new string("toto");
   delete pi;
   cout << (*(pi+4)) << endl;
}
```

```
jdeanton@FARCI:$./executable

jdeanton@FARCI:$
```

Attention `pi` peut exister sans
l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   string* pi = new string("toto");
   delete pi;
   cout << (*(pi+200)) << endl;
   *(pi+200) = "deantoni";
   cout << (*(pi+200)) << endl;
}
```

```
jdeanton@FARCI:$./executable
????
jdeanton@FARCI:$
```

Attention `pi` peut exister sans l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   string* pi = new string("toto");
   delete pi;
   cout << (*(pi+200)) << endl;
   *(pi+200) = "deantoni";
   cout << (*(pi+200)) << endl;
}
```

```
jdeanton@FARCI:$./executable

deantoni
jdeanton@FARCI:$
```

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
   delete pi;
   cout << (*pi) << endl;
}
```

```
jdeanton@FARCI:$./executable
0
jdeanton@FARCI:$
```

Attention `pi` peut exister sans
l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
   delete pi;
   pi=nullptr;
   cout << (*pi) << endl;
}
```

```
jdeanton@FARCI:$./executable
Segmentation fault (core dumped)
jdeanton@FARCI:$
```

Attention `pi` peut exister sans l'objet pointé…

→ **bonne pratique: pointeur a nullptr**

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   string* pi = new string(''toto'');
   delete pi;
   cout << (*(pi)) << endl;
}
```

```
jdeanton@FARCI:$./executable
Segmentation fault (core dumped)
jdeanton@FARCI:$
```

Attention `pi` peut exister sans l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   string* pi = new string(‘‘toto’’);
   delete pi;
   If (pi != nullptr){
     cout << (*(pi)) << endl;
   }
}
```

```
jdeanton@FARCI:$./executable
Segmentation fault (core dumped)
jdeanton@FARCI:$
```

Attention `pi` peut exister sans
l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   string* pi = new string(‘‘toto’’);
   delete pi;
   pi=nullptr;
   If (pi != nullptr){
     cout << (*(pi)) << endl;
   }
}
```

```
jdeanton@FARCI:$./executable
jdeanton@FARCI:$
```

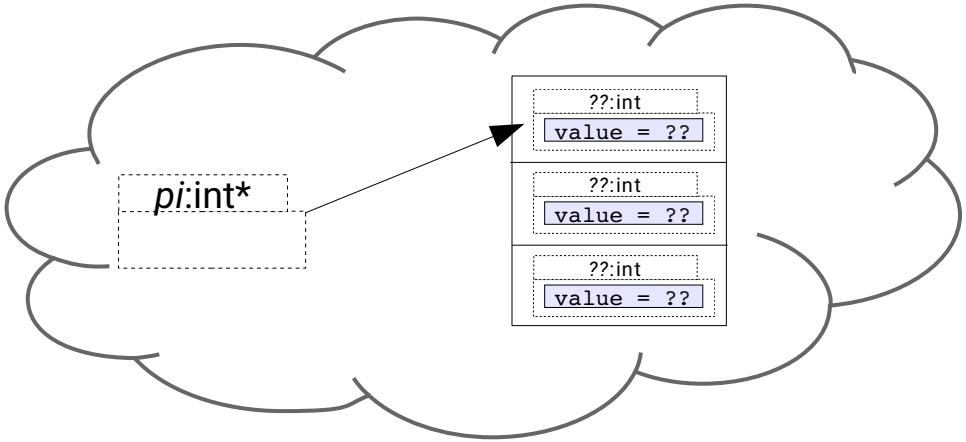
→ **bonne pratique: pointeur a nullptr**

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int[3];
}
```



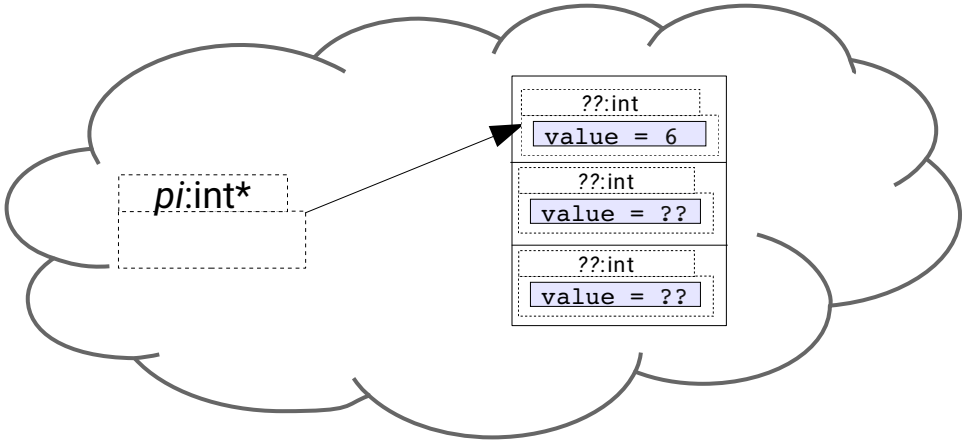
Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int[3];
   (*pi)=06;
   cout << (*pi) << endl;
}
```

```
jdeanton@FARCI:$./executable
6
jdeanton@FARCI:$
```



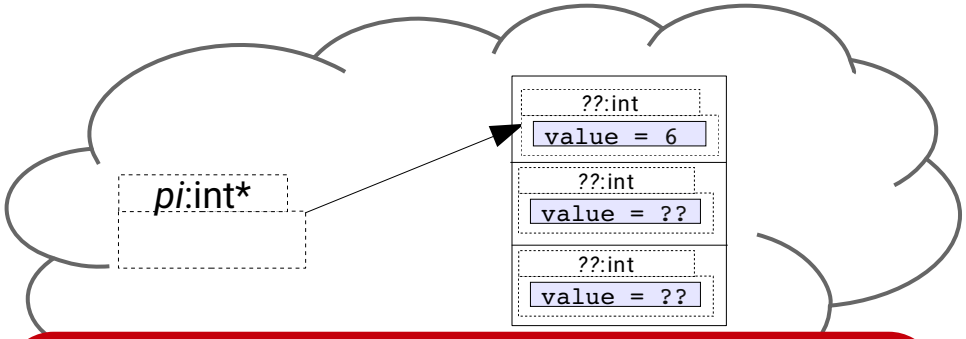
Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int[3];
   pi[0]=06;
   cout << (*pi) << endl;
}
```

```
jdeanton@FARCI:$./executable
6
jdeanton@FARCI:$
```



Note : accès à la valeur se trouvant à la case N d'un tableau

→ **pi[N] == *(pi + N * sizeof(Element))**

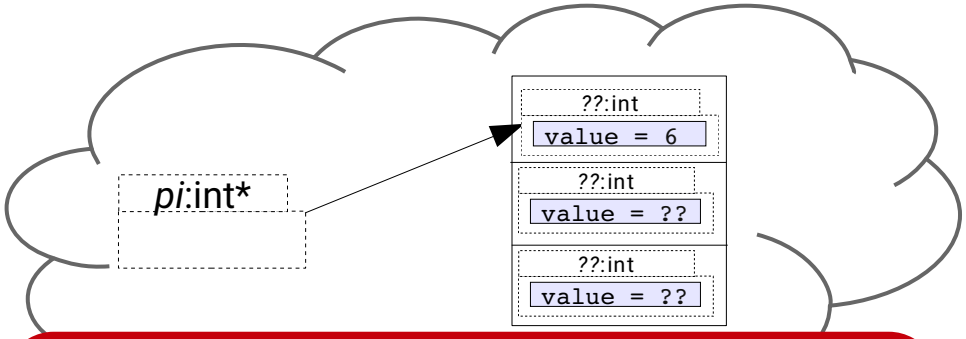
Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int[3];
   pi[12]=69;
   cout << pi[12] << endl;
}
```

```
jdeanton@FARCI:$./executable
????
jdeanton@FARCI:$
```



Note : accès à la valeur se trouvant à la case N d'un tableau

→ **pi[N] == *(pi + N * sizeof(Element))**

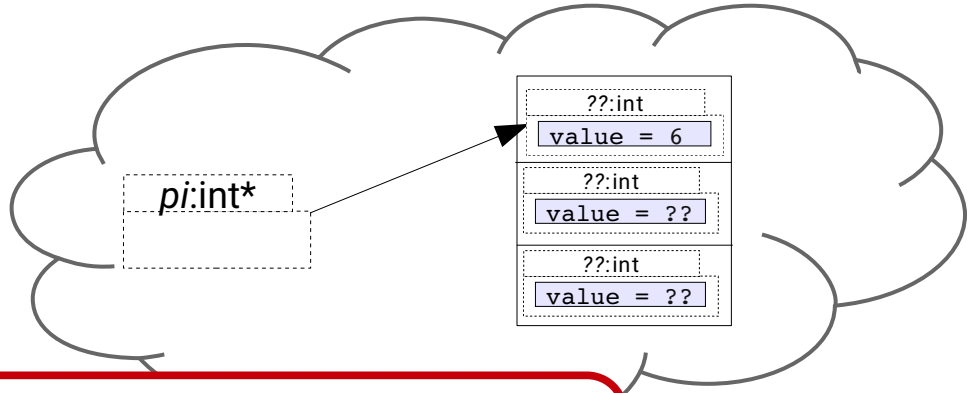
Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int[3];
   pi[12]=69;
   cout << pi[12] << endl;
}
```

```
jdeanton@FARCI:$./executable
69
jdeanton@FARCI:$
```



Éviter l'utilisation de tableau en C++ !!!

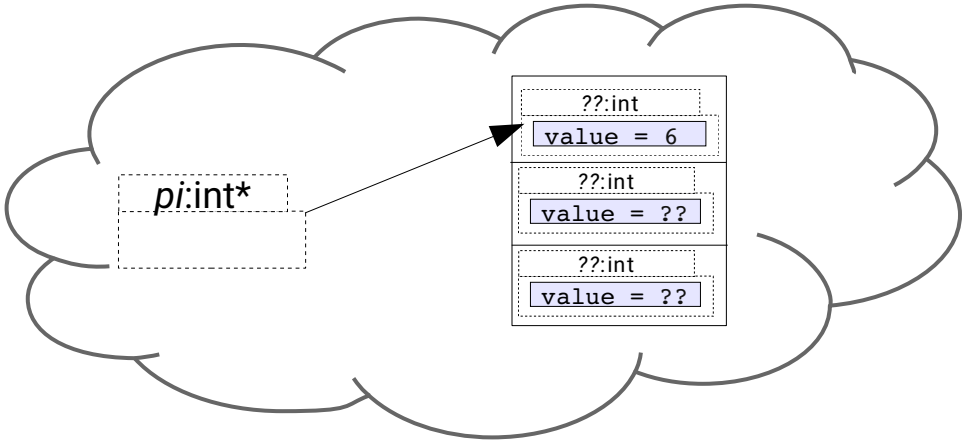
(utilisation des *containers* de la STL...)

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int[3];
   delete[] pi;
}
```

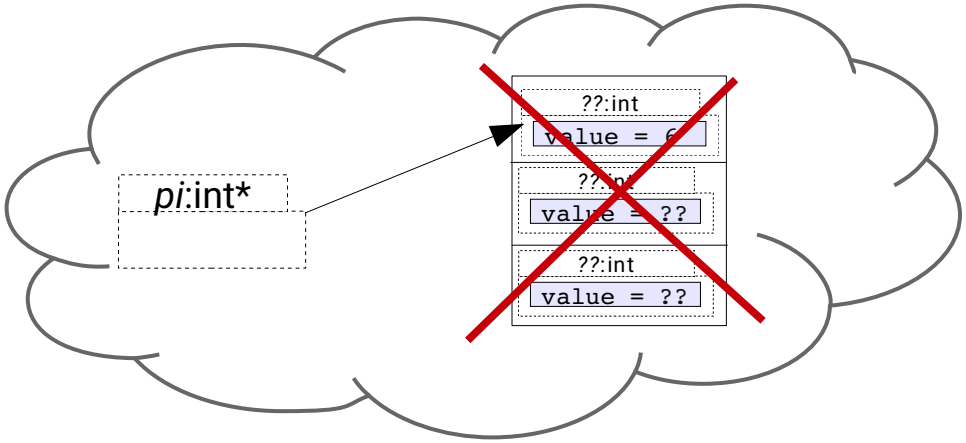


Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int[3];
   delete[] pi;
}
```

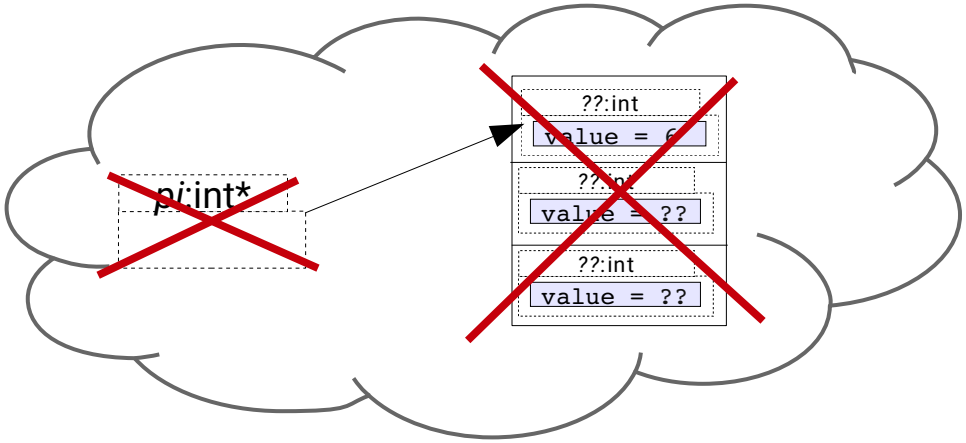


Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int[3];
   delete[] pi;
}
```

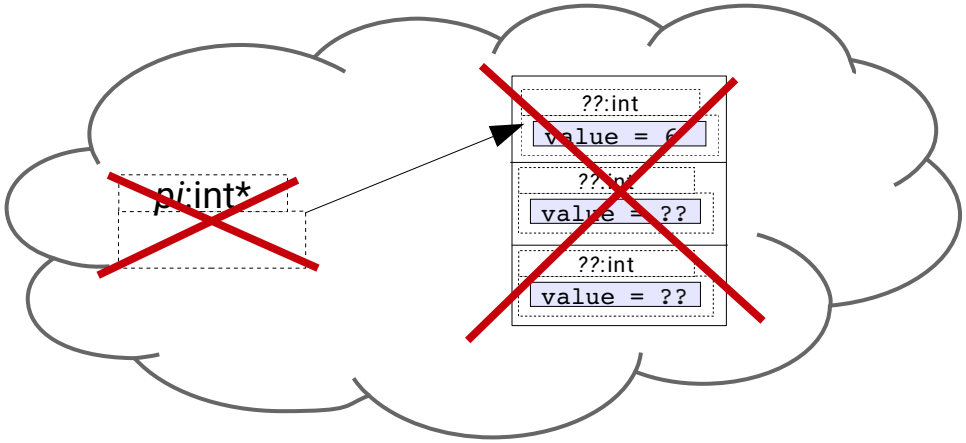


Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int[3];
   delete[] pi;
   pi = nullptr;
}
```



Plan

- Allocation dynamique de mémoire
- Les constructeurs / destructeurs
  - Constructeurs
    - Par défaut (synthétisé et explicite)
    - Transtypage
    - Transtypage + Paramètres par défaut
    - De copie (synthétisé et explicite)
  - Destructeurs

Constructeurs
intro

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle aRectangle;
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
                int length;
                int width;
  public:
                void resize(int nl, int nw);
                void scale(float factor);
};
#endif
```

rectangle.h

→ **que se passe t'il a la création d'un objet ?**

1. Création de la structure en mémoire
2. Appel implicite ou explicite du constructeur

Constructeurs
par défaut, synthétisé

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;
  Rectangle* r2 = new Rectangle();
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- Appel implicite ou explicite du constructeur
  - Un constructeur est une fonction membre généralement publique dont le nom est celui de la classe et pour laquelle il n'existe pas de type de retour.
  - Si aucun constructeur n'existe, **synthèse d'un constructeur par défaut** (sans paramètres)

Constructeurs
par défaut, synthétisé

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;
  Rectangle* r2 = new Rectangle();
  delete r2;
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- Appel implicite ou explicite du constructeur
  - Un constructeur est une fonction membre généralement publique dont le nom est celui de la classe et pour laquelle il n'existe pas de type de retour.
  - Si aucun constructeur n'existe, **synthèse d'un constructeur par défaut** (sans paramètres)

Constructeurs
par défaut, synthétisé

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle r1;
   Rectangle* pt_r2 = new Rectangle();
   delete pt_r2;
   pt_r2=nullptr;
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- Appel implicite ou explicite du constructeur
  - Un constructeur est une fonction membre généralement publique dont le nom est celui de la classe et pour laquelle il n'existe pas de type de retour.
  - Si aucun constructeur n'existe, synthèse d'un **constructeur par défaut** (sans paramètres)

Constructeurs
par défaut, synthétisé

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle r1;
   Rectangle* pt_r2 = new Rectangle();
   delete pt_r2;
   pt_r2=nullptr ;
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- Appel implicite ou explicite du constructeur
  - Un constructeur est une fonction membre généralement publique dont le nom est celui de la classe et pour laquelle il n'existe pas de type de retour.
  - Si aucun constructeur n'existe, synthèse d'un **constructeur par défaut** (sans paramètres)

Constructeurs
par défaut, synthétisé

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle r1;
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
                int length;
                int width;
  public:
                void resize(int nl, int nw);
                void scale(float factor);
};
#endif
```

rectangle.h

Si aucun constructeur n'existe,

- synthèse d'un **constructeur par défaut** (sans paramètres)
  - Initialisation « par défaut » des attributs
  - Appel des constructeurs par défaut si les attributs sont des objets

Constructeurs
par défaut, explicite

```
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle r1;

return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur par défaut** (sans paramètres) explicite
  - Initialisation des attributs
  - Appel des constructeurs des attributs si ce sont des objets

```
Rectangle::Rectangle(){
   length = 0;
   width = 0;
}
```

rectangle.cpp

Constructeurs par défaut, explicite

```
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;

return 0;
}
```

Lors de l'implémentation des fonctions membres d'une classe :

- Sa signature contient son espace de définition (de nommage, i.e., la classe)

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

```
Rectangle::Rectangle(){
  length = 0;
  width = 0;
}
```

rectangle.cpp

Constructeurs par défaut, explicite

```
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;
```

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

Lors de l'implémentation des fonctions membres d'une classe :

- Sa signature contient son espace de définition (de nommage, i.e., la classe)
- L'accès à l'objet représentant le paramètre caché se fait au travers du pointeur `this` (optionnel si pas de conflit)

```
Rectangle::Rectangle(){
  this-> length = 0;
  (*this).width = 0;
}
```

rectangle.cpp

Constructeurs
par défaut, explicite

```
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle r1;

return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur par défaut** (sans paramètres) explicite
  - Appel des constructeurs des attributs si ce sont des objets
  - Initialisation des attributs

```
Rectangle::Rectangle(){
   length = 0;
   width = 0;
}
```

rectangle.cpp

Constructeurs
par défaut, explicite

```
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;

return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur par défaut** (sans paramètres) explicite
  - Appel des « constructeurs » des attributs si ce sont des objets
  - Initialisation des attributs

```
Rectangle::Rectangle()
: length(), width(){
   length = 0;
   width = 0;
}
```

rectangle.cpp

par défaut, explicite

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;

return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur par défaut** (sans paramètres) explicite
  - Appel des « constructeurs » des attributs si ce sont des objets
  - Initialisation des attributs

initialisation

affectation

```cpp
Rectangle::Rectangle()
: length(), width(){
   length = 0;
   width = 0;
}
```

rectangle.cpp

Constructeurs
par défaut, explicite

```
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;

return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur par défaut** (sans paramètres) explicite
  - Appel des « constructeurs » des attributs si ce sont des objets
  - Initialisation des attributs

```
Rectangle::Rectangle(){
  length = 0;
  width = 0;
}
```

==

```
Rectangle::Rectangle()
: length(), width(){
  length = 0;
  width = 0;
}
```

rectangle.cpp

Appels aux « constructeurs » implicites !!

Constructeurs
par défaut, explicite

```
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;

return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur par défaut** (sans paramètres) explicite
  - Appel des « constructeurs » des attributs si ce sont des objets
  - Initialisation des attributs

```
Rectangle::Rectangle(){
  length = 0;
  width = 0;
}
```

!=

```
Rectangle::Rectangle()
: length(0), width(0){ }
```

rectangle.cpp

Appels aux « constructeurs » implicites !!

Constructeurs
par défaut, explicite

```
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;

return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            Segment s1,s2,s3,s4 ;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur par défaut** (sans paramètres) explicite
  - Appel des constructeurs des attributs si ce sont des objets
  - Initialisation des attributs

```
Rectangle::Rectangle()
: s1(10),s2(4),s3(10),s4(4)
{}
```

rectangle.cpp

Constructeurs
par défaut, explicite

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;

return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            Segment s1,s2,s3,s4 ;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

```cpp
#ifndef _SEGMENT_H_
#define _SEGMENT_H_

class Segment{
  [...]
  public:
            Segment(int);
  [...]
};
#endif
```

Segment.h

```cpp
Rectangle::Rectangle()
: s1(10),s2(4),s3(10),s4(4)
{}
```

rectangle.cpp

Constructeurs
par défaut, explicite

```
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle r1;

return 0;
}
```

Attention, le constructeur ne pourra pas être synthétisé

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            Segment s1,s2,s3,s4 ;
  public:
            // Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);

dif
```

rectangle.h

```
#ifndef _SEGMENT_H_
#define _SEGMENT_H_

class Segment{
  [...]
  public:
            Segment(int);
  [...]
};
#endif
```

Segment.h

```
Rectangle::Rectangle()
: s1(),s2(),s3(),s4()
{}
```

rectangle.cpp

Constructeurs
par défaut, explicite

```
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle r1;

return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur par défaut** (sans paramètres) explicite
  - Initialisation des attributs
  - Appel des constructeurs des attributs si ce sont des objets

```
Rectangle::Rectangle()
: length(0), width(0)
{}
```

rectangle.cpp

par défaut, explicite

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;

return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

C'est également de cette manière que l'on appelle les constructeurs des classes mères

```cpp
Rectangle::Rectangle()
: length(0), width(0)
{}
```

rectangle.cpp

Constructeurs
par défaut, explicite

C++11

```
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle r1;

return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
              int length = 0;
              int width = 0;
  public:

              void resize(int nl, int nw);
              void scale(float factor);
};
#endif
```

rectangle.h

À utiliser sans modération

Plan

- Allocation dynamique de mémoire
- Les constructeurs / destructeurs
  - Constructeurs
    - Par défaut (synthétisé et explicite)
    - **Transtypage**
    - Transtypage + Paramètres par défaut
    - De copie (synthétisé et explicite)
  - Destructeurs

Transtypage

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;            //ne compile plus
  Rectangle aRectangle(3);
  Rectangle r2 = 4 ;
  Rectangle r3 = {5} ;   //C++11
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle(int l);
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur de transtypage**
  - **construit un objet à partir d'un objet d'un autres type**
  - Initialisation des attributs
  - Appel des constructeurs des attributs si ce sont des objets

```cpp
Rectangle::Rectangle(int l){
  length = l;
  width = l;
}
```

rectangle.cpp

Transtypage

```
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;              //ne compile plus
  Rectangle aRectangle(3);
  Rectangle r2 = 4 ;
  Rectangle r3 = {5} ;   //C++11
return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle(int l);
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur de transtypage**
  - **construit un objet à partir d'un objet d'un autres type**
  - Initialisation des attributs
  - Appel des constructeurs des attributs si ce sont des objets

```
Rectangle::Rectangle(int l)
  :length(l), width(l) {  }
```

rectangle.cpp

Constructeurs

Transtypage

```
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;              //ne compile plus
  Rectangle aRectangle(3);
  Rectangle r2 = 4 ;
  Rectangle r3 = {5} ;   //C++11
  aRectangle.copyDimension(8) ;
return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle(int l);
            void resize(int nl, int nw);
            void copyDimension(Rectangle r);
};
#endif
```

rectangle.h

- **constructeur de transtypage**
  - **construit un objet à partir d'un objet d'un autres type**
  - Initialisation des attributs
  - Appel des constructeurs des attributs si ce sont des objets

Constructeurs

```
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;              //ne compile plus
  Rectangle aRectangle(3);
  Rectangle r2 = 4 ;
  Rectangle r3 = {5} ;   //C++11
  aRectangle.copyDimension(8) ;
return 0;
}
```

main.cpp

Transtypage

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle(int l);
            void resize(int nl, int nw);
            void copyDimension(Rectangle& r);
};
#endif
```

rectangle.h

```
jdeanton@linux-hani:~/boulot/enseignements/CPP/2021/cours/example_cours/Rectangle> make
g++ -g -c main.cpp -o main.o -I . -g -ansi -Wall -Wextra -Wold-style-cast -Woverloaded-virtual -D_DEBUG_ -std=c++11 -pthread
main.cpp: In function 'int main()':
main.cpp:16:30: error: cannot bind non-const lvalue reference of type 'Rectangle&' to an rvalue of type 'Rectangle'
   16 |     aRectangle.copyDimension(7);
      |                              ^
In file included from main.cpp:4:
Rectangle.h:13:5: note:   after user-defined conversion: 'Rectangle::Rectangle(int)'
   13 |     Rectangle(int l):length(l),width(l){}
      |     ^~~~~~~~~
Rectangle.h:16:35: note:   initializing argument 1 of 'void Rectangle::copyDimension(Rectangle&)'
   16 |     void copyDimension(Rectangle& r);
      |                        ~~~~~~~~~~~^
```

Constructeurs

Transtypage

```
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;            //ne compile plus
  Rectangle aRectangle(3);
  Rectangle r2 = 4 ;
  Rectangle r3 = {5} ;   //C++11
  aRectangle.copyDimension(8) ;
return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle(int l);
            void resize(int nl, int nw);
            void copyDimension(const Rectangle& r);
};
#endif
```

rectangle.h

More on left value versus right value here :
https://www.internalpointers.com/post/understanding-meaning-lvalues-and-rvalues-c

Constructeurs

Transtypage

```
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;              //ne compile plus
  Rectangle aRectangle= {3, 2}; //C++11

return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle(int l, int w);
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur** de transtypage
  - **construit un objet à partir d'objet(s) d'autres(s) type(s))**
    - Initialisation des attributs
  - Appel des constructeurs des attributs si ce sont des objets

```
Rectangle::Rectangle(int l, int w)
  :length(l), width(w) { }
```

rectangle.cpp

Plan

- Allocation dynamique de mémoire
- Les constructeurs / destructeurs
  - Constructeurs
    - Par défaut (synthétisé et explicite)
    - Transtypage
    - Transtypage + Paramètres par défaut
    - **De copie (synthétisé et explicite)**
  - Destructeurs

Constructeurs de copies explicite

```cpp
#include <iostream>
#include "rectangle.h"
void f(Rectangle r){
...
}
int main()
{
  Rectangle r1;
  Rectangle aRectangle = r1;
  Rectangle r2(r1);
  f(r2);
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle(const Rectangle& rec);
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- constructeur **explicite** (non synthetisé) de **copie**
  - construit un objet à partir d'un autre objet du même type
  - Prend une référence constante en paramètre

```cpp
Rectangle::Rectangle(const Rectangle& rec){
  length = rec.length;
  width = rec.width;
}
```

rectangle.cpp

Constructeurs de copies explicite

```cpp
#include <iostream>
#include "rectangle.h"
void f(Rectangle r){
...
}
int main()
{
   Rectangle r1;
   Rectangle aRectangle = r1;
   Rectangle r2(r1);
   f(r2);
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            Rectangle(const Rectangle& rec);
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- constructeur **explicite** (non synthetisé) de **copie**
  - construit un objet à partir d'un autre objet du même type
  - Prend une référence constante en paramètre

```cpp
Rectangle::Rectangle(const Rectangle& rec)
  :length(rec.length), width(rec.width) { }
```

rectangle.cpp

Constructeurs de copies explicite

```
#include <iostream>
#include "rectangle.h"
void f(Rectangle r){
...
}
int main()
{
   Rectangle r1;
   Rectangle aRectangle = r1;
   Rectangle r2(r1);
   f(r2);
return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Recta
  private:


  public:
            Rectangle(const Rectangle& rec);
            void resize(int nl, int nw);
            void scale(float factor);

};
#endif
```

rectangle.h

Pourquoi une référence constante ici ?

- constructeur **explicite** (non synthetisé) de **copie**
  - construit un objet à partir d'un autre objet du même type
  - Prend une référence constante en paramètre

```
Rectangle::Rectangle(const Rectangle& rec){
   length = rec.length;
   width = rec.width;
}
```

rectangle.cpp

Constructeurs de copies explicite

```cpp
#include <iostream>
#include "rectangle.h"
void f(Rectangle r){
...
}
int main()
{
  Rectangle r1;
  Rectangle aRectangle = r1;
  Rectangle r2(r1);
  f(r2);
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
        int length;
        int width;
  public:
        Rectangle(const Rectangle& rec);
        void resize(int nl, int nw);
        void scale(float factor);
};
#endif
```

rectangle.h

- constructeur **explicite** (non synthetisé) de **copie**
  - construit un objet à partir d'un autre objet du même type
  - Prend une référence constante en paramètre

```cpp
Rectangle::Rectangle(const Rectangle& rec){
  length = rec.length;
  width = rec.width;
}
```

rectangle.cpp

Constructeurs de copies explicite

```
#include <iostream>
#include "rectangle.h"
void f(Rectangle r){
...
}
int main()
{
   Rectangle r1;
   Rectangle aRectangl
   Rectangle r2(r1);
   f(r2);
return 0;
}
```

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
```

```
Rectangle& rec);
int nw);
actor);
```

rectangle.h

Le constructeur de copie est « *indispensable* » à l'utilisation correcte d'une classe

- constructeur expli
  - construit un objet à partir d'un autre objet du même type
  - Prend une référence constante en paramètre

```
Rectangle::Rectangle(const Rectangle& rec){
   length = rec.length;
   width = rec.width;
}
```

rectangle.cpp

Constructeurs de copies synthétisé

```cpp
#include <iostream>
#include "rectangle.h"
void f(Rectangle r){
...
}
int main()
{
  Rectangle r1;
  Rectangle aRectangle = r1;
  Rectangle r2(r1);
  f(r2);
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:

            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- constructeur **synthétisé** de **copie**
  - Copie de l'objet bit à bit (bitwise)
  - Utilise le constructeur par défaut des attributs

de copies synthétisé

- constructeur **synthétisé** de **copie**
  - Copie de l'objet bit à bit (bitwise)
  - Utilise le constructeur par défaut des attributs

```
#include <iostream>
#include "B.h"


class A{
  B* myB ;
} ;
```

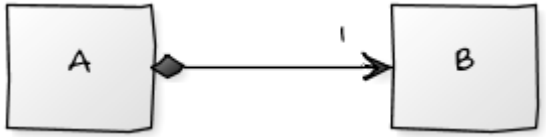
A.cpp

```
#include <iostream>
#include "B.h"


class A{
  B myB ;
} ;
```

A.cpp

de copies synthétisé

- constructeur **synthétisé** de **copie**
  - Copie de l'objet bit à bit (bitwise)
  - Utilise le constructeur par défaut des attributs




```
#include <iostream>
#include "B.h"


class A{
  B* myB ;
} ;
```

A.cpp

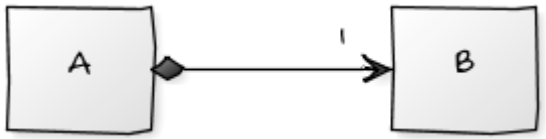
```
#include <iostream>
#include "B.h"


class A{
  B myB ;
} ;
```

A.cpp

de copies synthétisé

- constructeur **synthétisé** de **copie**
  - Copie de l'objet bit à bit (bitwise)
  - Utilise le constructeur par défaut des attributs

?




```
#include <iostream>
#include "B.h"


class A{
  B* myB ;
} ;
```

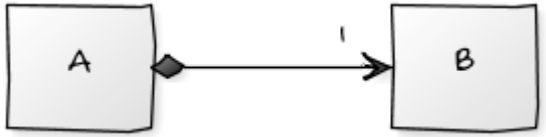
A.cpp

```
#include <iostream>
#include "B.h"


class A{
  B myB ;
} ;
```

A.cpp

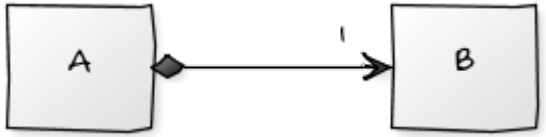
de copies synthétisé

- constructeur **synthétisé** de **copie**
  - Copie de l'objet bit à bit (bitwise)
  - Utilise le constructeur par défaut des attributs




```
#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A(){
    myB = new B() ;
  }
} ;
```

A.cpp

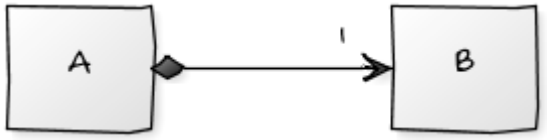



```
#include <iostream>
#include "B.h"


class A{
  B myB ;
} ;
```

A.cpp

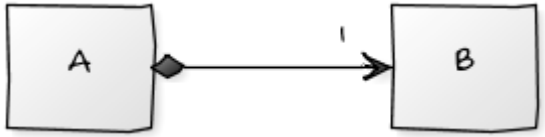
de copies synthétisé

- constructeur **synthétisé** de **copie**
  - Copie de l'objet bit à bit (bitwise)
  - Utilise le constructeur par défaut des attributs



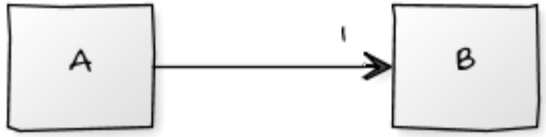
A.cpp

```
#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A(){
    myB = new B() ;
  }
} ;
```



A.cpp

```
#include <iostream>
#include "B.h"


class A{
  B myB ;
} ;
```



A.cpp

```
#include <iostream>
#include "B.h"

class A{
  B* myB ;

A(B& itsB){
    myB = &itsB ;
  }
} ;
```

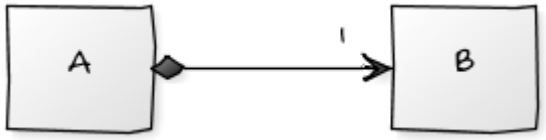
Constructeurs de copies synthétisé

Soyez sur de comprendre la différence entre les trois

constructeur synthétisé de copie

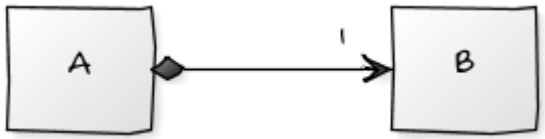
- Copie de l'objet bit à bit (bitwise)
- Utilise le constructeur par défaut des attributs

Dans quel cas le constructeur de copie synthétisé est-il suffisant ?



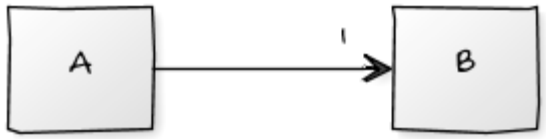

```
#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A(){
    myB = new B() ;
  }
} ;
```

A.cpp




```
#include <iostream>
#include "B.h"


class A{
  B myB ;
} ;
```

A.cpp




```
#include <iostream>
#include "B.h"

class A{
  B* myB ;

A(B& itsB){
    myB = &itsB ;
  }
} ;
```

A.cpp

Plan

- Allocation dynamique de mémoire
- Les constructeurs / destructeurs
  - Constructeurs
    - Par défaut (synthétisé et explicite)
    - Transtypage
    - **Paramètres par défaut**
    - De copie (synthétisé et explicite)
  - Destructeurs

Constructeurs
(Transtypage +) paramètres par défaut

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;
  Rectangle aRectangle(3,2);
  Rectangle r2 = 6 ;
  Rectangle r3 = {3}; //C++11
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
        int length;
        int width;
  public:
        Rectangle(int l=1, int w=1);
        void resize(int nl, int nw);
        void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur de transtypage**
  - construit un objet à partir d'objet(s) d'autres(s) type(s))
  - **Si appelé sans paramètres, les valeurs spécifiées par défaut sont utilisées (peut remplacer le constructeur par défaut)**

```cpp
Rectangle::Rectangle(int l, int w){
  length = l;
  width = w;
}
```

rectangle.cpp

Constructeurs
Transtypage + paramètres par défaut

Paramètres par défaut :

- Peuvent être utilisés dans tous types de fonctions (membres ou libres, constructeurs ou non)
- Liste terminale de paramètres par défaut
- La valeur par défaut n'est spécifiée que dans la déclaration (et pas répétée dans la définition)

```
int f(int i, double x=0.0, int j=1);
```

- Appel des constructeurs des attributs si ce sont

Constructeurs
Transtypage + paramètres par défaut

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
  Rectangle r1;
  Rectangle aRectangle(3,2);
  Rectangle r2 = 6 ;
  Rectangle r3 = {3}; //C++11
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
        int length;
        int width;
  public:
        Rectangle(int l=1, int w=1);
        void resize(int nl, int nw);
        void scale(float factor);
};
#endif
```

rectangle.h

- **constructeur de transtypage**
  - construit un objet à partir d'objet(s) d'autres(s) type(s))
  - **Si appelé sans paramètres, les valeurs spécifiées par défaut sont utilisées (peut remplacer le constructeur par défaut)**

```cpp
Rectangle::Rectangle(int l, int w){
  length = l;
  width = w;
}
```

rectangle.cpp

Plan

- Allocation dynamique de mémoire
- Les constructeurs / destructeurs
  - Constructeurs
    - Par défaut (synthétisé et explicite)
    - Transtypage
    - Transtypage + Paramètres par défaut
    - De copie (synthétisé et explicite)
  - **Destructeurs**

Destructeurs

intro

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle aRectangle;
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
   Private:
                int length;
                int width;
   Public:
                void resize(int nl, int nw);
                void scale(float factor);
};
#endif
```

rectangle.h

→ **que se passe t'il a la destruction d'un objet ?**
(statique ou dynamique)

1) Appel **implicite** du destructeur

- Lors de la fin d'un bloc pour les objets alloués statiquement
- Lors de l'appel à `delete` pour les objets alloués dynamiquement

2) Destruction de la structure en mémoire

Destructeurs

intro

```
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle aRectangle;
return 0;
}
```

main.cpp

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_
class Rectangle{
  Private:
            int length;
            int width;
  Public:
            ~Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

→ **que se passe t'il a la destruction d'un objet ?**
(statique ou dynamique)

- Un destructeur est une fonction membre **publique** dont le nom est celui de la classe précédé d'un tilde ('~ ')
- **Un destructeur ne prends jamais de paramètre**

Destructeurs synthétisé

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle aRectangle;
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_
class Rectangle{
  Private:
            int length;
            int width;
  Public:
            ~Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- Destruction de l'espace mémoire pour les attributs alloués statiquement....

Destructeurs
explicite

```cpp
#include <iostream>
#include "rectangle.h"

int main()
{
   Rectangle aRectangle;
return 0;
}
```

main.cpp

```cpp
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

class Rectangle{
  private:
            int length;
            int width;
  public:
            ~Rectangle();
            void resize(int nl, int nw);
            void scale(float factor);
};
#endif
```

rectangle.h

- Permet de libérer la mémoire allouée dynamiquement pour l'objet en question.
- Permet de détruire d'autres objets au travers de l'operateur `delete`

```cpp
Rectangle::~Rectangle(){
//rien a faire. Le synthétisé suffisait
}
```

rectangle.cpp

Destructeurs explicite

```
#ifndef _A_H_
#define _A_H_

#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A() ;
  ~A() ;

} ;
#endif
```

A.h

```
A::A(){
    myB = new B() ;
 }

A::~A(){
   delete myB ;
}
```

A.cpp

Destructeurs
explicite

```
#ifndef _A_H_
#define _A_H_

#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A() ;
  ~A() ;

} ;
#endif
```

A.h

```
A::A(){
    myB = new B() ;
 }

A::~A(){
   delete myB ;
}
```

A.cpp

Cette classe est incorrecte,
Pourquoi ?

Destructeurs
explicite

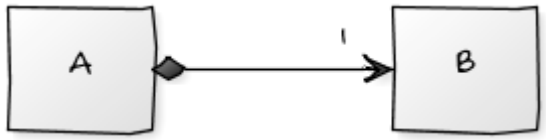
```
#ifndef _A_H_
#define _A_H_

#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A() ;
  ~A() ;

} ;
#endif
```

A.h

```
A::A(){
    myB = new B() ;
 }

A::~A(){
   delete myB ;
}
```

A.cpp

Cette classe est incorrecte,
Pourquoi ?



Destructeurs
explicite

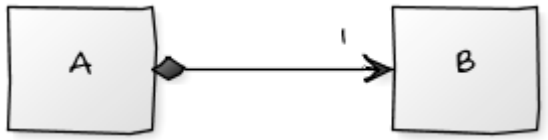
```
#ifndef _A_H_
#define _A_H_

#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A() ;
  ~A() ;

} ;
#endif
```

A.h

```
A::A(){
    myB = new B() ;
 }

A::~A(){
   delete myB ;
}
```

A.cpp

Cette classe est incorrecte,
Pourquoi ?



Le constructeur de copie synthétisé ne respecte pas la notion de *« contenance »*

Destructeurs
explicite

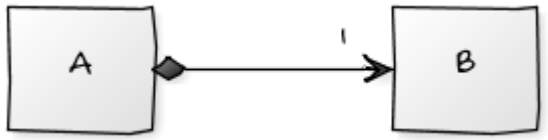
```
#ifndef _A_H_
#define _A_H_

#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A() ;
  A(const A& a) ;
  ~A() ;

} ;
#endif
```

A.h

```
A::A(){
    myB = new B() ;
}

A::A(const A& a){
   myB = new B(*(a.myB)) ;
}

A::~A(){
   delete myB
}
```

A.cpp

Cette classe est toujours incorrecte,
Pourquoi ?



Le constructeur de copie synthétisé ne respecte pas la notion de *« contenance »*

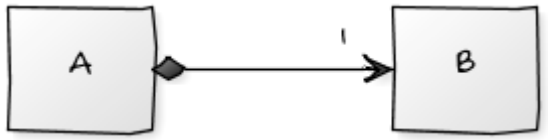
Destructeurs explicite

```
#ifndef _A_H_
#define _A_H_

#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A() ;
  A(const A& a) ;
  ~A() ;

} ;
#endif
```

A.h

```
A::A(){
    myB = new B() ;
}
A::A(const A& a){
   if (a.myB != nullptr){
      myB = new B(*(a.myB)) ;
   }
}

A::~A(){
   delete myB
}
```

A.cpp



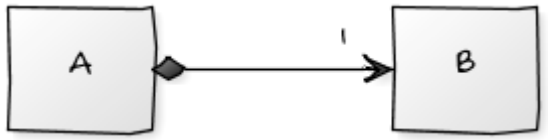
Destructeurs
explicite

```
#ifndef _A_H_
#define _A_H_

#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A() ;
  A(const A& a) ;
  ~A() ;

} ;
#endif
```

A.h

```
A::A(){
    myB = new B() ;
}
A::A(const A& a){
   if (a.myB != nullptr){
      myB = new B(*(a.myB)) ;
   }
}

A::~A(){
   delete myB
}
```

A.cpp



Cette classe est toujours incorrecte,
Pourquoi ?

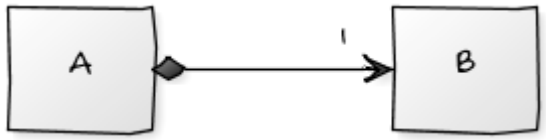
Destructeurs
explicite

```
#ifndef _A_H_
#define _A_H_

#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A() ;
  A(const A& a) ;
  ~A() ;

} ;
#endif
```

A.h

```
A::A(){
    myB = new B() ;
}
A::A(const A& a){
   if(a.myB != nullptr){
      myB = new B(*(a.myB)) ;
   }else{
      myB = nullptr ;
   }
}

A::~A(){
   delete myB
}
```

A.cpp



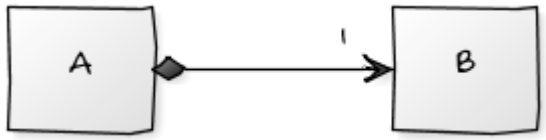
Destructeurs explicite

```
#ifndef _A_H_
#define _A_H_

#include <iostream>
#include "B.h"

class A{
  B* myB ;

  A() ;
  A(const A& a) ;
  ~A() ;

} ;
#endif
```

A.h

```
A::A(){
    myB = new B() ;
}
A::A(const A& a){
   if(a.myB != nullptr){
      myB = new B(*(a.myB)) ;
   }else{
      myB = nullptr ;
   }
}

A::~A(){
   delete myB
}
```

A.cpp



Et l'opérateur d'affectation ?..