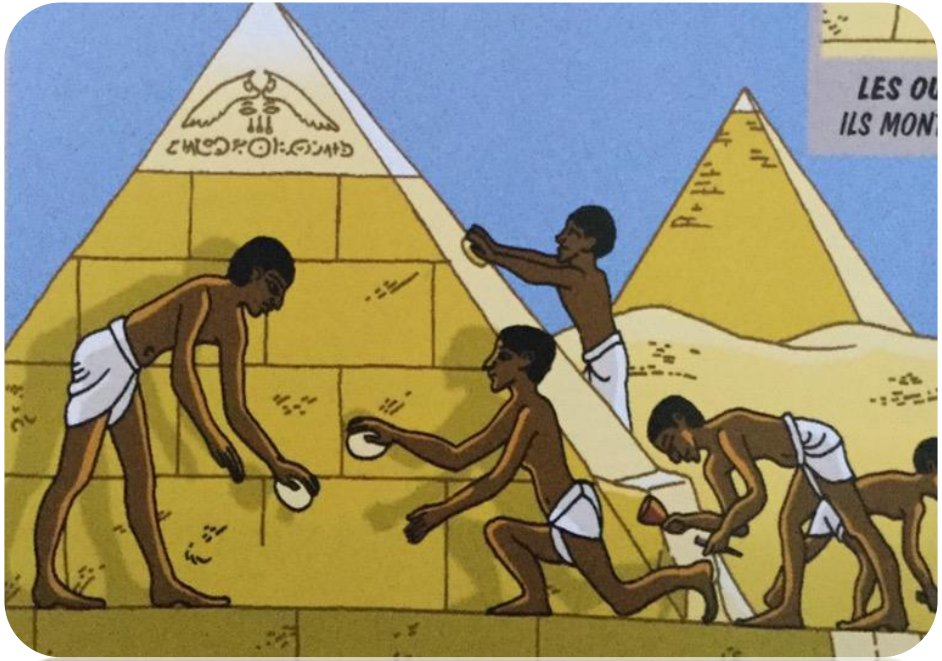

# Programmation orientée objet en C++

## - Constructeur et Destructeur -

**Groupe étudiants : CIR2**

Nils Beaussé
E-Mail : nils.beausse@isen-ouest.yncrea.fr
Numéro de bureau : A2-78




# Les Constructeurs

# Initialisation d'une instance de classe

- **Première solution** : affecter individuellement une valeur à chaque attribut via :

  - Les accesseurs :

```
Etudiant etd1;
etd1.setDevoir(15);
etd1.setExam(13);
```

  - Les attributs (directement) s'il sont déclarés publiques

```
Etudiant etd1;
etd1.devoir=15;
etd1.exam=13;
```

# Initialisation d'une instance de classe

- **Deuxième solution** : définir une méthode dédiée à l'initialisation des attributs

```
class Etudiant{
private:
  double devoir;
  double exam;
public:
  void init (double d, double e){
              devoir=d;
              exam=e;
    }
…
};
```

```
Etudiant etd1;
etd1.init(15,13)
```

# Initialisation d'une instance de classe

- **Deuxième solution** : définir une méthode dédiée à l'initialisation des attributs

```
class Etudiant{
private:
  double devoir;
  double exam;
public:
  void init (double d, double e){
              devoir=d;
              exam=e;
    }
…
};
```

```
Etudiant etd1;
etd1.init(15,13)
```

**Très proche du principe des constructeurs**

# Initialisation d'une instance de classe

- **Troisième solution** : **utilisation des constructeurs**

# Initialisation d'une instance de classe

- **Troisième solution** : **utilisation des constructeurs**

- Un **constructeur** est une méthode particulière :
  - **Invoquée automatiquement** lors de la déclaration d'un objet
  - publique
  - dont **son nom est le même que la classe**
  - sans type de retour (même pas void)

# Initialisation d'une instance de classe

- **Troisième solution** : **utilisation des constructeurs**

- Un **constructeur** est une méthode particulière :
  - **Invoquée automatiquement** lors de la déclaration d'un objet
  - publique
  - dont **son nom est le même que la classe**
  - sans type de retour (même pas void)

- On peut avoir plusieurs constructeurs dans une classe

```
class MyClass{
private:
 ...
public:
 MyClass(); // constructeur par défaut
 MyClass(type1 arg1, …);
 MyClass(type2 arg2, …);
};
```

- Chaque classe a un constructeur par défaut → généré automatiquement si aucun constructeur n'existe

# Initialisation d'une instance de classe

Exemple :

```
class bonhomme{
private:
 int poids;
 bool moche;
public:
bonhomme() // constructeur par défaut
 {
   moche = true;
   poids = 25;
 };
};
```

```
int main()
{
   bonhomme pierre;
}
```

Après ce point :
**pierre.poids = 25** et **moche = true**

# Initialisation d'une instance de classe

Exemple :

```
class bonhomme{
private:
 int poids;
 bool moche;
public:
bonhomme() // constructeur par défaut
 {
   moche = true;
   poids = 25;
 };
};
```

```
int main()
{
   bonhomme pierre;
}
```

Après ce point :
**pierre.poids = 25** et **moche = true**

OK, mais on aurait pu le mettre ici, tout simplement

# Initialisation d'une instance de classe

Exemple :

```
class bonhomme{
private:
 int poids = 25;
 bool moche = true;
public:
bonhomme() // constructeur par défaut
 {
   moche = true;
   poids = 25;
 };
};
```

```
int main()
{
   bonhomme pierre;
}
```

Après ce point :
**pierre.poids = 25** et **moche = true**

OK, mais on aurait pu le mettre ici, tout simplement

# Initialisation d'une instance de classe

Exemple :

```
class bonhomme{
private:
 int poids = 25;
 bool moche = true;
public:
bonhomme() // constructeur
 {
   moche = true;
   poids = 25;
 };
};
```

```
pierre;
```

**Mais alors à quoi ça sert ?**

point :
**pierre.poids = 25** et **moche = true**

OK, mais on aurait pu le mettre ici, tout simplement

# Initialisation d'une instance de classe

Exemple plus compliqué :

```
class bonhomme{
private:
 int poids;
 bool moche;
public:
bonhomme() // constructeur par défaut
 {
   string reponse;
   bool valide = 0;
   poids = 25;
   do
   {
     cout << "Le bonhomme est-il moche ?" << endl;
     cin >> reponse;
     valide = true;

     if (reponse == "oui")        moche = 1;
     else if (reponse == "non" ) moche = 0;
     else valide = false;

   }while (valide == false);

 };
};
```

```
int main()
{
bonhomme pierre;
cout << boolalpha;
cout << "le bonhomme est moche : " << pierre.getmoche() << endl;
}
```

Résultat :

```
Le bonhomme est-il moche ?
peut-être
Le bonhomme est-il moche ?
maisheuuu
Le bonhomme est-il moche ?
oui
le bonhomme est moche : true
```

# Déclaration d'un constructeur

- Syntaxe :

**MyClass.h**

```
#ifndef _Etudiant_H
#define _Etudiant_H

class MyClass{
private:
 ...
public:
 MyClass(); // constructeur par défaut
 MyClass(type1 arg1, …);
 MyClass(type2 arg2, …);
 ...
};

#endif
```

**MyClass.cpp**

```
#include "Etudiant.h"

MyClass::MyClass() {
...
      }
MyClass::MyClass(type1 arg1, …){
...
}
MyClass::MyClass(type2 arg2, …){
      ...
      }
...
```

# Déclaration d'un constructeur

- Exemple :

**Etudiant.h**

```
#ifndef _Etudiant_H
#define _Etudiant_H

class Etudiant{
private:
  double devoir;
  double exam;
Public:
  ...
  Etudiant();
  Etudiant(double d);
  Etudiant(double d, double e);
};

#endif
```

**Etudiant.cpp**

```
#include "Etudiant.h"

Etudiant::Etudiant() {
setDevoir(1);
setExam(0);
      }
Etudiant::Etudiant(double d){
setDevoir(d);
setExam(0);
}
Etudiant::Etudiant(double d, double e){
      setDevoir(d);
setExam(e);
      }
...
```

# Appel des constructeurs

- **Allocation statique**

  - **Syntaxe** :

```
Nom_classe nom_instance(arg1, arg2, …); // ou sans argument
```

  - **Exemple** :

```
Etudiant etd1(12,13);
```

- **Allocation dynamique**

  - **Syntaxe** :

```
Nom_classe* nom_instance=new Nom_classe(arg1, arg2, …); // ou sans argument
```

  - **Exemple** :

```
Etudiant* etd1=new Etudiant(12,13);
```

# Appel des constructeurs

- **Allocation statique**

  - **Syntaxe** :

```
Nom_classe nom_instance(arg1, arg2, …); // ou sans argument
```

  - **Exemple** :

```
Etudiant etd1(12,13);
```

- **Allocation dynamique**

  - **Syntaxe** :

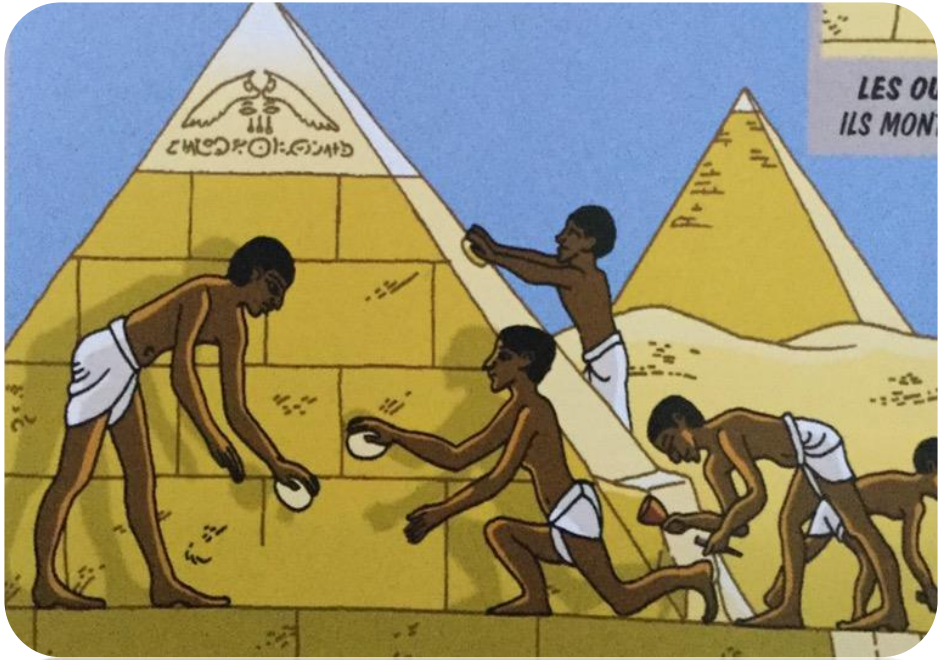
```
Nom_classe* nom_instance=new Nom_classe(arg1, arg2, …); // ou sans argument
```

  - **Exemple** :

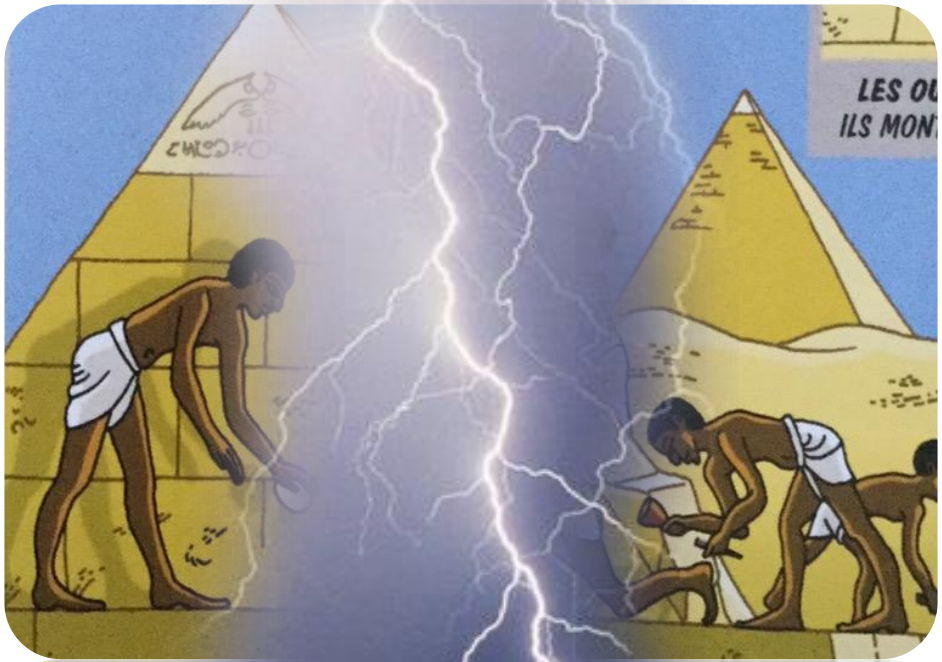
```
Etudiant* etd1=new Etudiant(12,13);
```

**Contrairement au malloc du C, le new appelle automatiquement le constructeur !**

**C'est pour cette raison qu'on doit utiliser new en C++, car il y beaucoup d'objet avec des constructeurs !**




# Les Constructeurs




# SURCHARGÉS !!!!

# Surcharge du constructeur

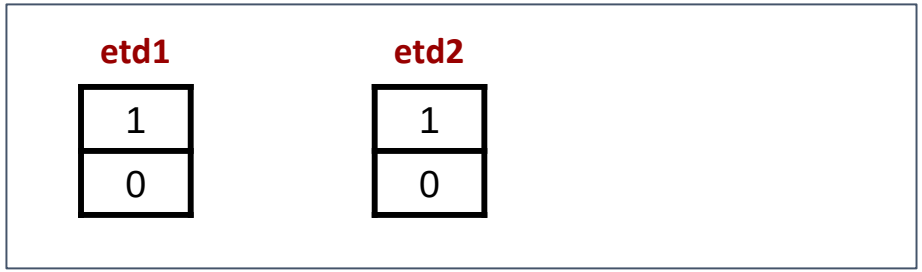
- La version surchargée appropriée du constructeur est appelée en se basant sur :
  - **Le nombre, le type et l'ordre des arguments**

- Exemple :
  - Appel implicite :

```
Etudiant etd1;
Etudiant* etd2 = new Etudiant
```

  - Appel explicite :

```
Etudiant* etd2 = new Etudiant();
```




**Etudiant.cpp**

```
#include "Etudiant.h"

Etudiant::Etudiant() {
setDevoir(1);
setExam(0);
      }
Etudiant::Etudiant(double d){
setDevoir(d);
setExam(0);
}
Etudiant::Etudiant(double d, double e){
      setDevoir(d);
setExam(e);
      }
...
```

# Surcharge du constructeur

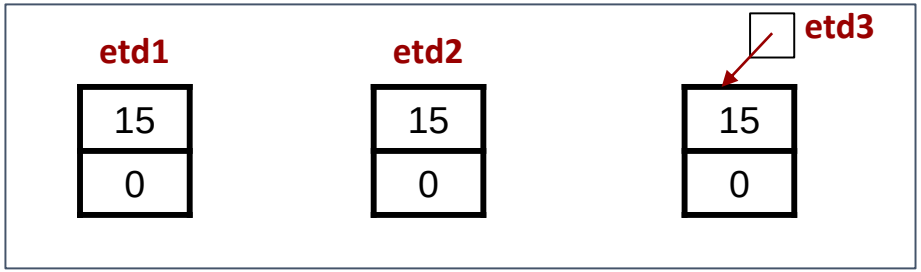
- La version surchargée appropriée du constructeur est appelée en se basant sur :
  - **Le nombre, le type et l'ordre des arguments**

- Exemple :
  - Appel implicite :

```
Etudiant etd1(15);
```

  - Appel explicite :

```
Etudiant etd2 = Etudiant(15);
Etudiant* etd3 = new Etudiant(15);
```




**Etudiant.cpp**

```
#include "Etudiant.h"

Etudiant::Etudiant() {
setDevoir(1);
setExam(0);
      }
Etudiant::Etudiant(double d){
setDevoir(d);
setExam(0);
}
Etudiant::Etudiant(double d, double e){
      setDevoir(d);
setExam(e);
      }
...
```

# Surcharge du constructeur

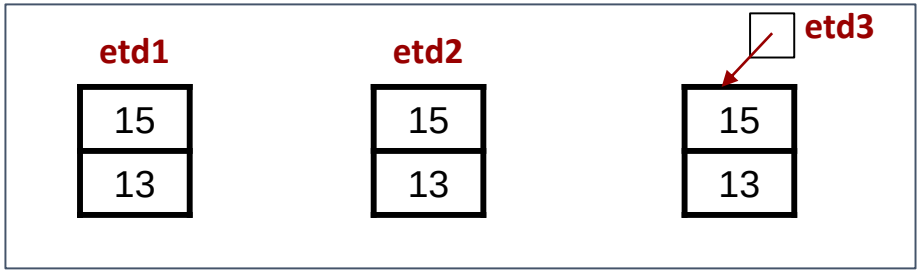
- La version surchargée appropriée du constructeur est appelée en se basant sur :
  - **Le nombre, le type et l'ordre des arguments**

- Exemple :
  - Appel implicite :

```
Etudiant etd1(15,13);
```

  - Appel explicite :

```
Etudiant etd2 = Etudiant(15,13);
Etudiant* etd3 = new Etudiant(15,13);
```




**Etudiant.cpp**

```
#include "Etudiant.h"

Etudiant::Etudiant() {
setDevoir(1);
setExam(0);
      }
Etudiant::Etudiant(double d){
setDevoir(d);
setExam(0);
}
Etudiant::Etudiant(double d, double e){
      setDevoir(d);
setExam(e);
      }
...
```

# Surcharge du constructeur

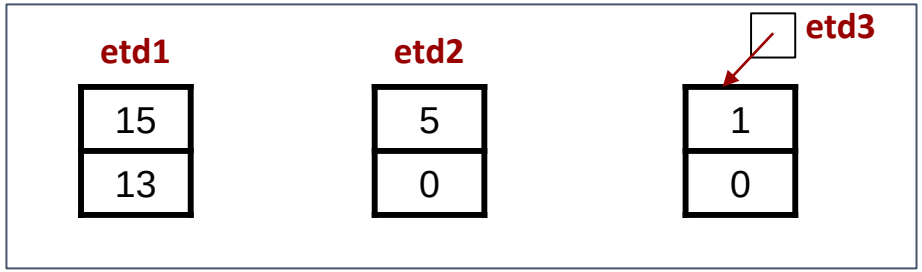
- Une bonne solution pour une gestion optimale des objets ->
- Exemple :

- Appel implicite :

```
Etudiant etd1(15,13);
```

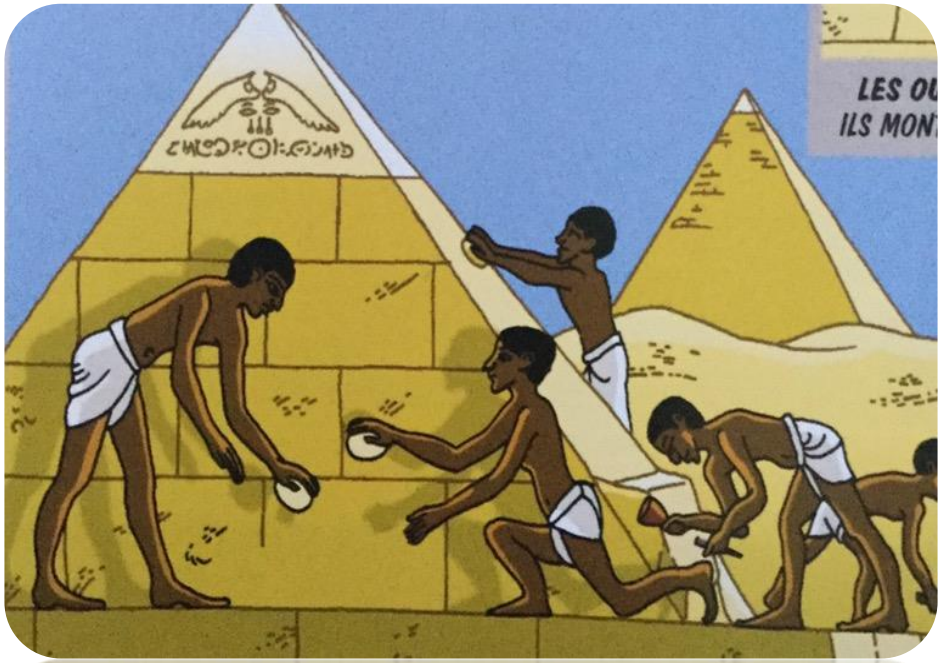
- Appel explicite :

```
Etudiant etd2 = Etudiant(5);
Etudiant* etd3 = new Etudiant();
```




**Etudiant.cpp**

```
#include "Etudiant.h"

Etudiant::Etudiant() {
setDevoir(1);
setExam(0);
      }
Etudiant::Etudiant(double d){
setDevoir(d);
setExam(0);
}
Etudiant::Etudiant(double d=1, double e=0){
      setDevoir(d);
setExam(e);
      }
...
```




# Les Constructeurs

## Liste d'initialisation et constructeurs par défaut

# Liste d'initialisation

- **Objectif :** une autre façon d'initialiser les attributs de classe : plus propre pour le compilateur, notamment.

**Sans liste d'initialisation**

```
MyClass(type1 arg1, type2 arg2, ..){
    arg1 = val1;
    arg2 = val2;
    ...
}
```

**Avec une liste d'initialisation**

```
MyClass(type1 arg1, type2 arg2, ..) : arg1(val1), arg2(val2){
    ...
}
```

- **Exemple :**

**Sans liste d'initialisation**

```
Etudiant(double d, double e){
    devoir = d;
    exam = e;
    ...
}
```

**Avec une liste d'initialisation**

```
Etudiant(double d, double e) : devoir(d), exam(e){
    ...
}
```

**Avec une liste d'initialisation**

```
Etudiant(double devoir, double exam) : devoir(devoir), exam(exam){
    // pas de masquage
}
```

# Constructeur par défaut

- Un constructeur par défaut est un constructeur sans arguments/paramètres ou qui a une valeur par défaut
- Exemple :
  - Appel implicite :

```
Etudiant etd1;
```

sans ()


**Etudiant.cpp**

```
#include "Etudiant.h"

Etudiant::Etudiant() {
setDevoir(1);
setExam(0);
      }
Etudiant::Etudiant(double d){
setDevoir(d);
setExam(0);
}
Etudiant::Etudiant(double d, double e){
      setDevoir(d);
setExam(e);
      }
...
```

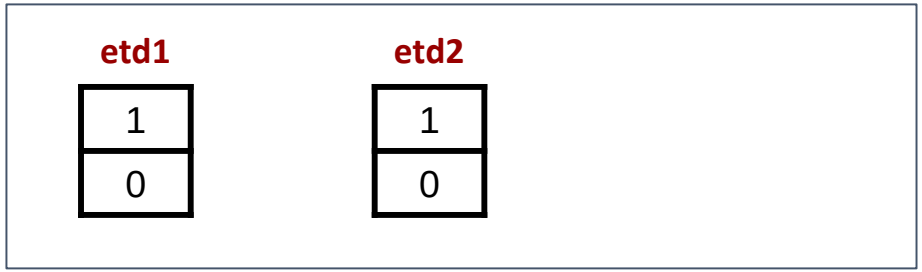
# Constructeur par défaut

- Un constructeur par défaut est un constructeur sans arguments/paramètres ou qui a une valeur par défaut
- Exemple :
  - Appel implicite :

```
Etudiant etd1;
```

  - Appel explicite :

```
Etudiant etd2 = Etudiant();
```




**Etudiant.cpp**

```
#include "Etudiant.h"

Etudiant::Etudiant() {
setDevoir(1);
setExam(0);
      }
Etudiant::Etudiant(double d){
setDevoir(d);
setExam(0);
}
Etudiant::Etudiant(double d, double e){
      setDevoir(d);
setExam(e);
      }
...
```

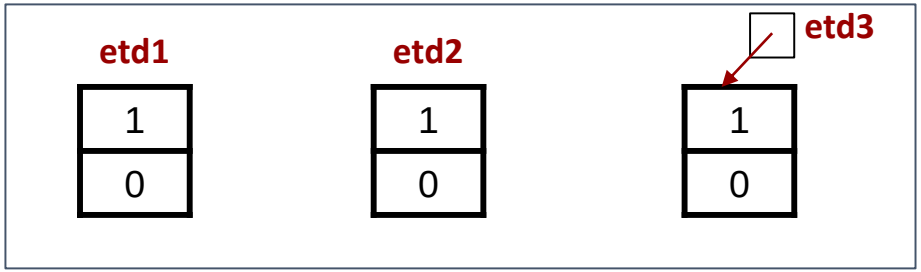
# Constructeur par défaut

- Un constructeur par défaut est un constructeur sans arguments/paramètres ou qui a une valeur par défaut
- Exemple :
  - Appel implicite :

```
Etudiant etd1;
```

  - Appel explicite :

```
Etudiant etd2 = Etudiant();
Etudiant* etd3 = new Etudiant();
```




**Etudiant.cpp**

```
#include "Etudiant.h"

Etudiant::Etudiant() {
setDevoir(1);
setExam(0);
      }
Etudiant::Etudiant(double d){
setDevoir(d);
setExam(0);
}
Etudiant::Etudiant(double d, double e){
      setDevoir(d);
setExam(e);
      }
...
```

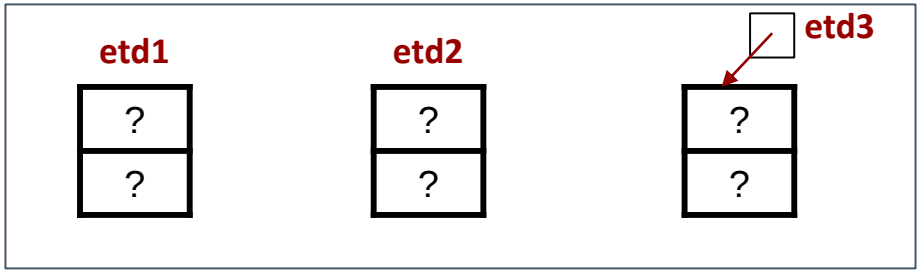
# Constructeur par défaut par défaut

- Si aucun constructeur n'est spécifié explicitement, le compilateur génère automatiquement une version minimale du constructeur par défaut qui :
  - appelle le constructeur par défaut des attributs objets
  - laisse non initialisé les attributs de type de base
- Exemple :
  - Appel implicite :

```
Etudiant etd1;
```

  - Appel explicite :

```
Etudiant etd2 = Etudiant();
Etudiant* etd3 = new Etudiant();
```




Etudiant.cpp

```
#include "Etudiant.h"

Etudiant::Etudiant() {
setDevoir(1);
setExam(0);
      }
Etudiant::Etudiant(double d){
setDevoir(d);
setExam(0);
}
Etudiant::Etudiant(double d, double e){
      setDevoir(d);
setExam(e);
      }
...
```

# Constructeur par défaut par défaut

- Si aucun constructeur n'est spécifié explicitement, le compilateur génère automatiquement une version minimale du constructeur par défaut qui :
  - appelle le constructeur par défaut des attributs objets
  - laisse non initialisé les attributs de type de base
- Exemple :
  - Appel implicite :

```
Etudiant etd1;
```

  - Appel explicite :

```
Etudiant etd2 = Etudiant();
Etudiant* etd3 = new Etudiant();
```

Etudiant.cpp

```
#include "Etudiant.h"

Etudiant::Etudiant() {
setDevoir(1);
setExam(0);
      }
Etudiant::Etudiant(double d){
setDevoir(d);
setExam(0);
}
Etudiant::Etudiant(double d, double e){
      setDevoir(d);
setExam(e);
      }
...
```

# Constructeur par défaut par défaut

- Si aucun constructeur n'est spécifié explicitement, le compilateur génère automatiquement une version minimale du constructeur par défaut qui :
  - appelle le constructeur par défaut des attributs objets
  - laisse non initialisé les attributs de type de base
- Exemple :
  - Appel implicite :

```
Etudiant etd1;
```

  - Appel explicite :

```
Etudiant etd2 = Etudiant();
Etudiant* etd3 = new Etudiant();
```

**Erreur de compilation :**

```
error: no matching function for
call to Etudiant::Etudiant()
```

**Etudiant.cpp**

```
#include "Etudiant.h"

Etudiant::Etudiant() {
setDevoir(1);
setExam(0);
      }
Etudiant::Etudiant(double d){
setDevoir(d);
setExam(0);
}
Etudiant::Etudiant(double d, double e){
      setDevoir(d);
setExam(e);
      }
...
```

# Constructeur par défaut

| | Constructeur par défaut | Etudiant e1; | Etudiant e1(15,16); |
|---|---|---|---|
| (A) | | | |
| (B) | | | |
| (C) | | | |
| (D) | | | |

(A)

```
Class Etudiant {
private:
 double exam;
 double devoir;
};
```

# Constructeur par défaut

| | Constructeur par défaut | Etudiant e1; | Etudiant e1(15,16); |
|---|---|---|---|
| **(A)** | constructeur par défaut par défaut | ? ? | Erreur de compilation |
| **(B)** | | | |
| **(C)** | | | |
| **(D)** | | | |

**(A)**

```
Class Etudiant {
private:
 double exam;
 double devoir;
};
```

# Constructeur par défaut

| | Constructeur par défaut | Etudiant e1; | Etudiant e1(15,16); |
|---|---|---|---|
| **(A)** | constructeur par défaut par défaut | ? ? | Erreur de compilation |
| **(B)** | | | |
| **(C)** | | | |
| **(D)** | | | |

**(B)**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant() {
devoir = 1;
Exam = 0;
      }
};
```

**ou**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant() : devoir(1), exam(0){}
};
```

# Constructeur par défaut

| | Constructeur par défaut | Etudiant e1; | Etudiant e1(15,16); |
|---|---|---|---|
| **(A)** | constructeur par défaut par défaut | ? ? | Erreur de compilation |
| **(B)** | Constructeur par défaut | 1 0 | Erreur de compilation |
| **(C)** | | | |
| **(D)** | | | |

**(B)**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant() {
devoir = 1;
Exam = 0;
      }
};
```

**ou**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant() : devoir(1), exam(0){}
};
```

# Constructeur par défaut

| | Constructeur par défaut | Etudiant e1; | Etudiant e1(15,16); |
|---|---|---|---|
| **(A)** | constructeur par défaut par défaut | ? ? | Erreur de compilation |
| **(B)** | Constructeur par défaut | 1 0 | Erreur de compilation |
| **(C)** | | | |
| **(D)** | | | |

**(C)**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant(double devoir=1, double exam=0) {
this -> devoir = devoir;
this -> exam = exam;
      }
};
```

**ou**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant(double devoir=1, double exam=0) : devoir(devoir), exam(exam){}
};
```

# Constructeur par défaut

| | Constructeur par défaut | Etudiant e1; | Etudiant e1(15,16); |
|---|---|---|---|
| **(A)** | constructeur par défaut par défaut | ? ? | Erreur de compilation |
| **(B)** | Constructeur par défaut | 1 0 | Erreur de compilation |
| **(C)** | | | |
| **(D)** | | | |

**(C)**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant(double devoir=1, double exam=0) {
this -> devoir = devoir;
this -> exam = exam;
      }
};
```

**ou**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant(double devoir=1, double exam=0) : devoir(devoir), exam(exam){}
};
```

**Pas de masquage !**

# Constructeur par défaut

| | Constructeur par défaut | Etudiant e1; | Etudiant e1(15,16); |
|---|---|---|---|
| **(A)** | constructeur par défaut par défaut | ? ? | Erreur de compilation |
| **(B)** | Constructeur par défaut | 1 0 | Erreur de compilation |
| **(C)** | Constructeur par défaut | 1 0 | 15 16 |
| **(D)** | | | |

**(C)**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant(double devoir=1, double exam=0) {
this -> devoir = devoir;
this -> exam = exam;
      }
};
```

**ou**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant(double devoir=1, double exam=0) : devoir(devoir), exam(exam){}
};
```

**Pas de masquage !**

# Constructeur par défaut

| | Constructeur par défaut | Etudiant e1; | Etudiant e1(15,16); |
|---|---|---|---|
| **(A)** | constructeur par défaut par défaut | ? ? | Erreur de compilation |
| **(B)** | Constructeur par défaut | 1 0 | Erreur de compilation |
| **(C)** | Constructeur par défaut | 1 0 | 15 16 |
| **(D)** | | | |

**(D)**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant(double devoir, double exam) {
this -> devoir = devoir;
this -> exam = exam;
      }
};
```

**ou**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant(double devoir, double exam) : devoir(devoir), exam(exam){}
};
```

# Constructeur par défaut

| | Constructeur par défaut | Etudiant e1; | Etudiant e1(15,16); |
|---|---|---|---|
| **(A)** | constructeur par défaut par défaut | ? ? | Erreur de compilation |
| **(B)** | Constructeur par défaut | 1 0 | Erreur de compilation |
| **(C)** | Constructeur par défaut | 1 0 | 15 16 |
| **(D)** | Pas de constructeur par défaut | Erreur de compilation | 15 16 |

**(D)**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant(double devoir, double exam) {
this -> devoir = devoir;
this -> exam = exam;
      }
};
```

**ou**

```
Class Etudiant {
private:
 double exam; double devoir;
public:
Etudiant(double devoir, double exam) : devoir(devoir), exam(exam){}
};
```

# Constructeur par défaut

| | |
|---|---|
| | );|
| (A) | tion |
| (B) | tion |
| (C) | |
| (D) | |

**À retenir :** la création d'un constructeur par l'utilisateur détruit le constructeur « *par défaut par défaut* » crée de base par le compilateur pour tout les objets.

Donc si on créé un constructeur **`Etudiant(double devoir, double exam)`** le constructeur standard **`Etudiant()`** est détruit. Si on ne le construit pas nous même, le code :

```
Etudiant paul;
```

**Ne compile plus !**

(D)

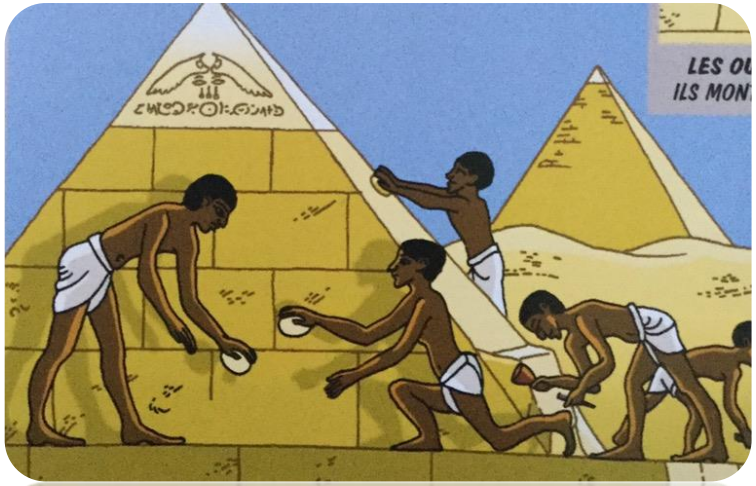
```
Clas
priv
 dou
publ
Etudi
this -> devoir = devoir;
this -> exam = exam;
      }
};
```

```
Etudiant(double devoir, double exam) : devoir(devoir), exam(exam){}
};
```

# Appel aux autres constructeurs

- C++11 autorise les constructeurs d'une classe à appeler n'importe quel autre constructeur de cette même classe
- Exemple :

```
Class Etudiant {
private:
 double exam; double devoir;
public:
 Etudiant(double devoir, double exam) : devoir(devoir), exam(exam){}
 Etudiant() : Etudiant(1,0) {}
};
```




# Les Constructeurs

# Les Constructeurs

## DE COPIE !

# Constructeur de copie

- Un constructeur de copie permet d'initialiser une instance à partir d'une autre instance
- Un constructeur de copie est appelé lorsqu'on:
  - **initialise une instance à partir d'une instance existante**

```
Etudiant e1(15,16);

Etudiant e2 = e1; // appel implicite
Etudiant e3(e1); // appel implicite
Etudiant e4 = Etudiant(e1); // appel explicite
```

# Constructeur de copie

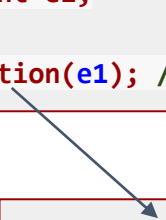
- Un constructeur de copie permet d'initialiser une instance à partir d'une autre instance
- Un constructeur de copie est appelé lorsqu'on:
  - **initialise une instance à partir d'une instance existante**

```
Etudiant e1(15,16);

Etudiant e2 = e1; // appel implicite
Etudiant e3(e1); // appel implicite
Etudiant e4 = Etudiant(e1); // appel explicite
```

# Constructeur de copie

- Un constructeur de copie permet d'initialiser une instance à partir d'une autre instance
- Un constructeur de copie est appelé lorsqu'on:
  - initialise une instance à partir d'une instance existante
  - **passe une instance par valeur comme argument d'une fonction**

```
Etudiant e1;

mafonction(e1); // appel implicite
```



```
void mafonction(Etudiant e2); // prototype de la fonction
```

Car rappel : un passage dans une fonction est un passage par copie !

# Constructeur de copie

- Un constructeur de copie permet d'initialiser une instance à partir d'une autre instance
- Un constructeur de copie est appelé lorsqu'on:
  - initialise une instance à partir d'une instance existante
  - passe une instance par valeur comme argument d'une méthode
  - **retourne une instance locale de l'objet dans une fonction**

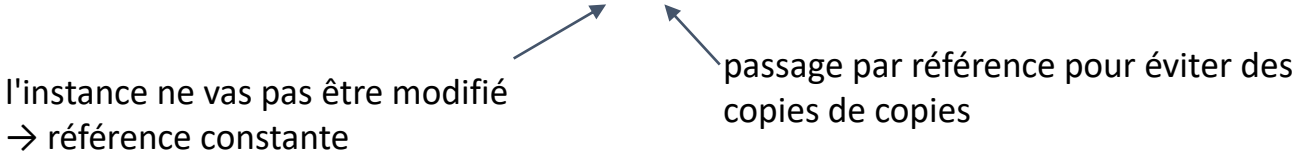
```
Etudiant mafonction(){
      Etudiant e1;
      ...
      return e1;
}
```

**Car ici, quand on return, e1 est copié dans la valeur de retour !**

# Constructeur de copie

- Syntaxe :

```
Nom_classe(NomClasse const& autre){..}
```



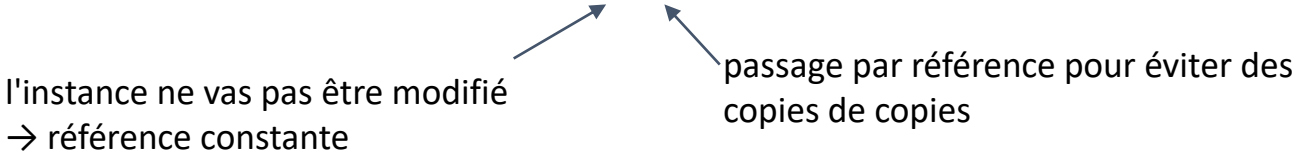

- Exemple :

```
Etudiant(Etudiant const& autre_etu)
{
   devoir = autre_etu.devoir;
   exam = autre_etu.exam;
}
```

# Constructeur de copie

- Syntaxe :

```
Nom_classe(NomClasse const& autre){..}
```




- Exemple (**variante avec liste d'init**) :

```
Etudiant(Etudiant const& autre) : devoir(autre.devoir), exam(autre.exam){}
```

# Constructeur de copie

- Syntaxe :

```
Nom_classe(NomClasse const& autre){..}
```

l'instance ne vas pas être modifié
→ référence constante

passage par référence pour éviter des copies de copies

- Exemple (variante avec liste d'init) :

```
Etudiant(Etudiant const& autre) : devoir(autre.devoir), exam(autre.exam){}
```

```
// appel implicite du constructeur de copie
Etudiant e1(15,16);
Etudiant e3=e1;
```

```
// appel implicite du constructeur de copie
Etudiant e1(15,16);
Etudiant e3(e1);
```

# Constructeur de copie

- Syntax

Attention à la confusion entre « int a = » et « a = »

Créer un constructeur de copie permet :

```
Etudiant e3=e1;
```

Mais il n'est pas appelé pour :

```
Etudiant e3;
e3=e1;
```

éviter des

- Exemp

```
m(autre.exam){}
```

```
// appel implicite du constructeur de copie
Etudiant e1(15,16);
Etudiant e3=e1;
```

```
// appel implicite du constructeur de copie
Etudiant e1(15,16);
Etudiant e3(e1);
```

# Constructeur de copie

- Syntax

Attention à la confusion entre « int a = » et « a = »

Créer un constructeur de copie permet :

```
Etudiant e3=e1;
```

Mais il n'est pas appelé pour :

```
Etudiant e3;
e3=e1;
```

→ Appelle le constructeur normal

éviter des

- Exemp

```
m(autre.exam){}
```

```
// appel implicite du constructeur de copie
Etudiant e1(15,16);
Etudiant e3=e1;
```

```
// appel implicite du constructeur de copie
Etudiant e1(15,16);
Etudiant e3(e1);
```

# Constructeur de copie

- Syntax

Attention à la confusion entre « int a = » et « a = »

Créer un constructeur de copie permet :

```
Etudiant e3=e1;
```

Mais il n'est pas appelé pour :

```
Etudiant e3;
e3=e1;
```

→ Appelle le constructeur normal

→ Appelle le système de copie par défaut

éviter des

- Exemp

```
m(autre.exam){}
```

```
// appel implicite du constructeur de copie
Etudiant e1(15,16);
Etudiant e3=e1;
```

```
// appel implicite du constructeur de copie
Etudiant e1(15,16);
Etudiant e3(e1);
```

# Constructeur de copie

- Syntax

- Exemp

Attention à la confusion entre « int a = » et « a = »

Créer un constructeur de copie permet :

```
Etudiant e3=e1;
```

Mais il n'est pas appelé pour :

```
Etudiant e3;
e3=e1;
```

→ Appelle le constructeur normal

→ Appelle le système de copie par défaut

**On verra plus tard comment changer le système lié au signe « = » en dehors des initialisations !**

```
// appel implicite du constructeur de copie
Etudiant e1(15,16);
Etudiant e3=e1;
```

```
// appel implicite du constructeur de copie
Etudiant e1(15,16);
Etudiant e3(e1);
```

# Constructeur de copie par défaut

- Un constructeur de copie est automatiquement généré par le compilateur s'il n'est pas explicitement défini = constructeur de copie par défaut

Par défaut :

```
Class Etudiant {
private:
 double exam;
 double devoir;
public :
Etudiant(double devoir=1, double exam=0) : devoir(devoir), exam(exam){}

};
```

```
int main()
{
Etudiant paul(3,2);
Etudiant jacque=paul;
}
```

**Fonctionne !**

# Constructeur de copie par défaut

- Un constructeur de copie est automatiquement généré par le compilateur s'il n'est pas explicitement défini = constructeur de copie par défaut

Si des constructeurs de copie par défauts existent, alors pourquoi en définir nous même ??

# Constructeur de copie par défaut

- Un constructeur de copie est automatiquement généré par le compilateur s'il n'est pas explicitement défini = constructeur de copie par défaut
- **Exemple de cas plus compliqué :**

```
Class Etudiant {
private:
   float* tableau_de_note;
public :
Etudiant(float note_1 = 10)
{
 tableau_de_note = new float[25];
 tableau_de_note[0] = note_1;
}
};
```

```
int main()
{
Etudiant paul(12.5);
Etudiant jacque=paul;
}
```

# Constructeur de copie par défaut

- Un constructeur de copie est automatiquement généré par le compilateur s'il n'est pas explicitement défini = constructeur de copie par défaut
- **Exemple de cas plus compliqué :**

```
Class Etudiant {
private:
   float* tableau_de_note;
public :
Etudiant(float note_1 = 10)
{
 tableau_de_note = new float[25];
 tableau_de_note[0] = note_1;
}
};
```

```
int main()
{
Etudiant paul(12.5);
Etudiant jacque=paul;
}
```

**=> GROSSE ERREUR**

**On va copier le pointeur tableau_de_note, donc l'adresse qui est pointé et non son contenu !**

**Les pointeurs vont se retrouver à pointer vers la même zone mémoire !**

**Alors qu'en fait on voulait que les deux objets aient chacun un tableau de note et aient les mêmes notes dans les deux tableaux.**

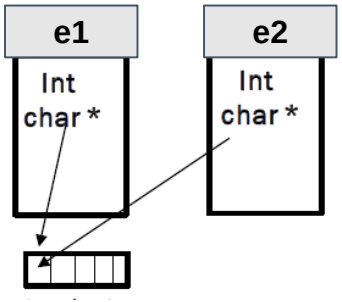
# Constructeur de copie par défaut

- Un constructeur de copie est automatiquement généré par le compilateur s'il n'est pas explicitement défini = constructeur de copie par défaut

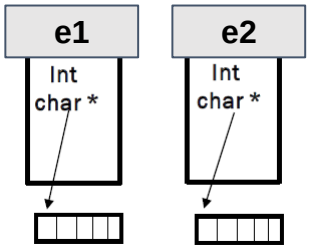
- Règle :

> Un constructeur de copie DOIT être défini lorsqu'une classe contient des attributs alloués dynamiquement (pointeurs)

```
Etudiant e2(e1);
```




**Utilisation du constructeur de copie par défaut**




**Utilisation d'un constructeur de copie défini par le programmeur**

# Constructeur de copie

- Qu'affiche ce code ?

```
class Person{
private:
char* name = nullptr;
public:
Person(char*);
void setName(char*);
void showName();
void print();
void replaceCharInName(char, char);
};
```

```
int main()
{
	Person p1("Smith");
	Person p2 = p1;
	p1.print();
	p2.print();
	cout << endl;
	p1.replaceCharInName('i', 'I');
	p1.print();
	p2.print();
	return 0;
}
```

# Constructeur de copie

- Qu'affiche ce code ?

```
class Person{
private:
char* name = nullptr;
public:
Person(char*);
void setName(char*);
void showName();
void print();
void replaceCharInName(char, char);
};
```

```
int main()
{
	Person p1("Smith");
	Person p2 = p1;
	p1.print();
	p2.print();
	cout << endl;
	p1.replaceCharInName('i', 'I');
	p1.print();
	p2.print();
	return 0;
}
```

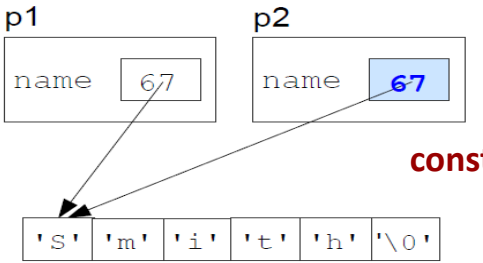
```
Smith
Smith

SmIth
SmIth
```

# Constructeur de copie

- Qu'affiche ce code ?

```
class Person{
private:
char* name = nullptr;
public:
Person(char*);
void setName(char*);
void showName();
void print();
void replaceCharInName(char, char);
};
```

```
int main()
{
	Person p1("Smith");
	Person p2 = p1;
	p1.print();
	p2.print();
	cout << endl;
	p1.replaceCharInName('i', 'I');
	p1.print();
	p2.print();
	return 0;
}
```



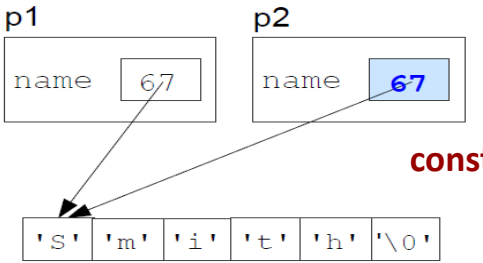

**Utilisation du constructeur de copie par défaut**

# Constructeur de copie

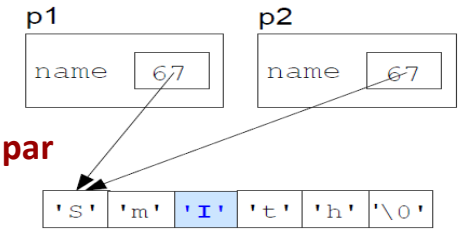
- Qu'affiche ce code ?

```
class Person{
private:
char* name = nullptr;
public:
Person(char*);
void setName(char*);
void showName();
void print();
void replaceCharInName(char, char);
};
```

```
int main()
{
	Person p1("Smith");
	Person p2 = p1;
	p1.print();
	p2.print();
	cout << endl;
	p1.replaceCharInName('i', 'I');
	p1.print();
	p2.print();
	return 0;
}
```

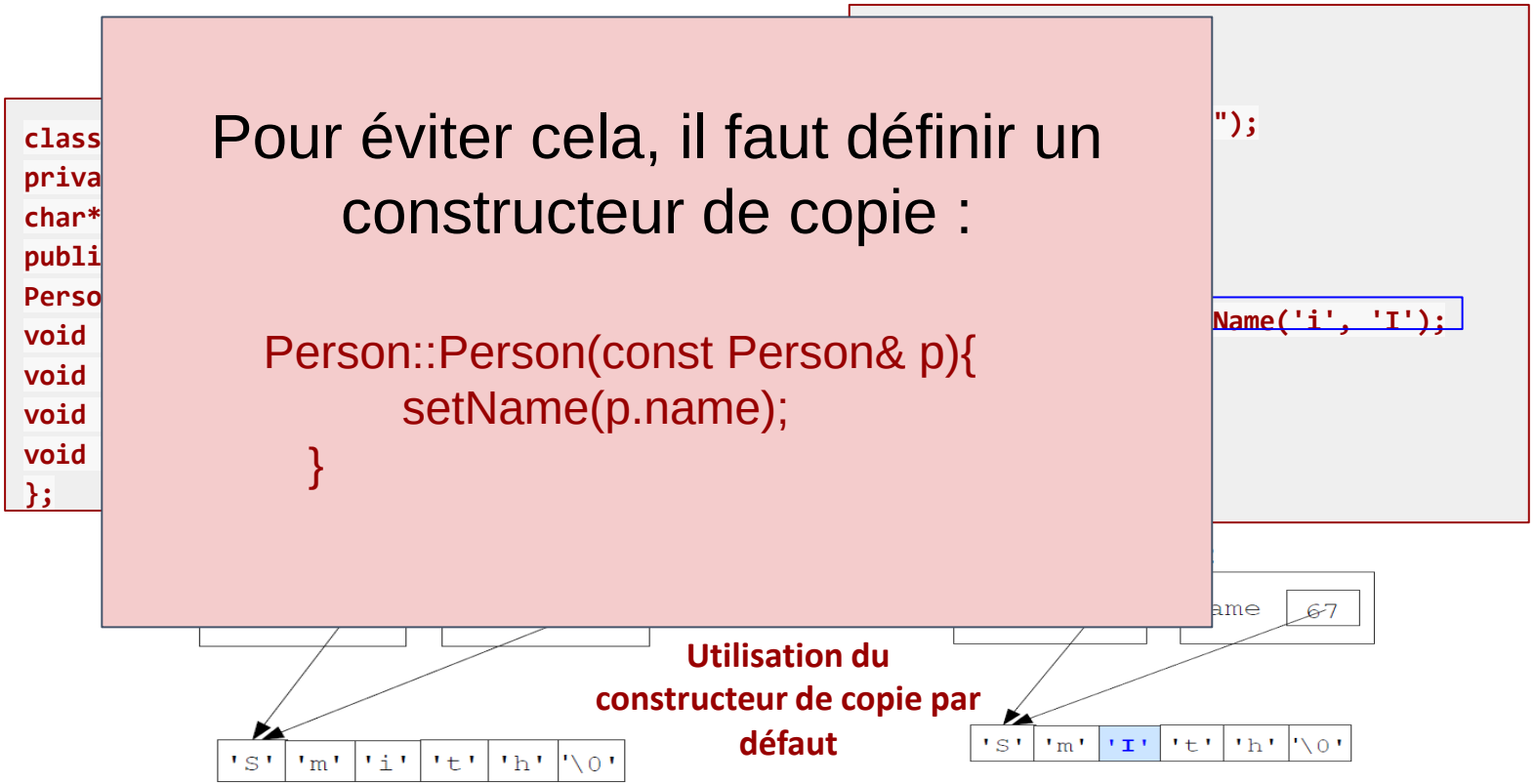



**Utilisation du constructeur de copie par défaut**



# Constructeur de copie

- Qu'affiche ce code ?



# Constructeur de copie

- Le constructeur de copie n'est pas appelé dans les cas suivants :
  - quand l'opérateur = n'est pas utilisé dans le contexte de la déclaration des variables

```
Etudiant e1(15,16);
Etudiant e2;
e2 = e1;
```

# Constructeur de copie

- Le constructeur de copie n'est pas appelé dans les cas suivants :
  - quand l'opérateur = n'est pas utilisé dans le contexte de la déclaration des variables
  - quand les références ou les pointeurs sont utilisés

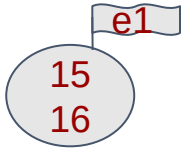
```
Etudiant e1(15,16);
Etudiant& e2 = e1;
```

```
Etudiant e1(15,16);
Etudiant* e2 = &e1;
```

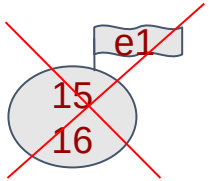
# Les Destructeurs

# Début et fin d'un objet

- **Constructeur** : Début de vie d'un objet




- **Destructeur** : Fin de vie d'un objet




- C'est quoi l'intérêt de détruire un objet ?

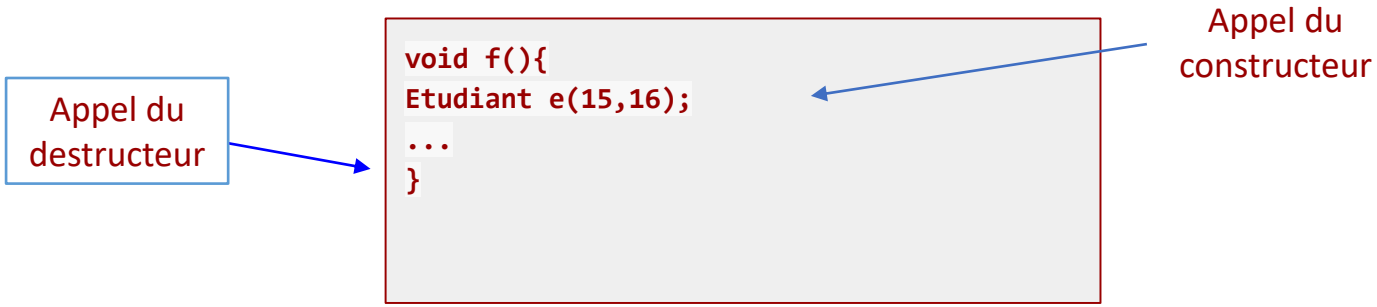
Si l'initialisation des attributs d'une instance implique la mobilisation de ressources :

- Fichiers
- Périphériques
- Portions de mémoire
- ...

Il est important de libérer ces ressources après usage !

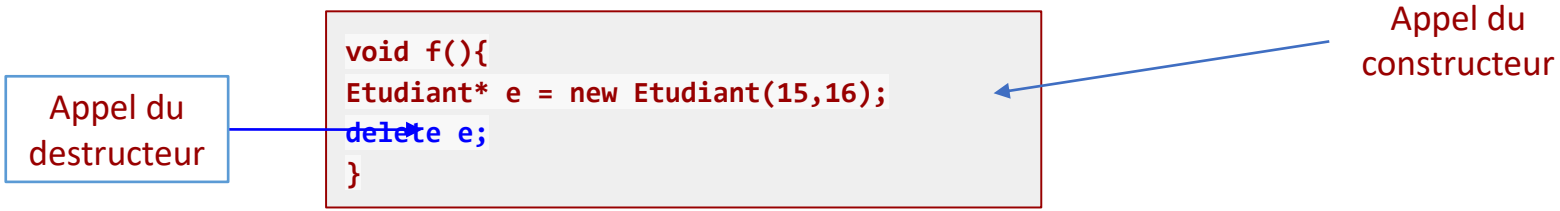
# Appel de destructeurs

- Le destructeur est appelé :
  - **à la fin du bloc si l'instance a été déclarée statiquement :**



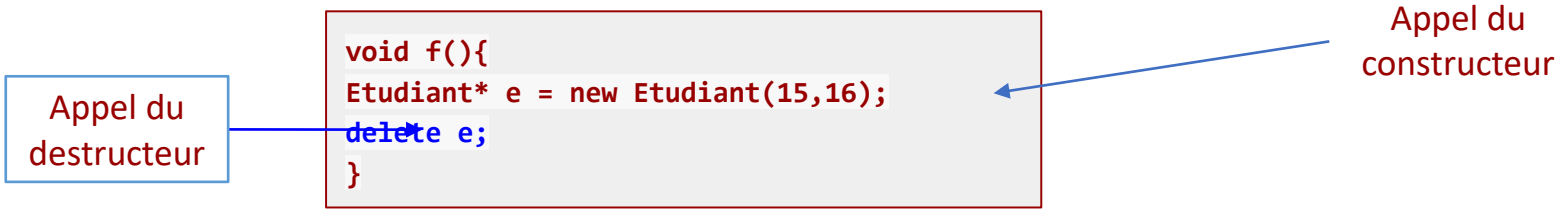
# Appel de destructeurs

- Le destructeur est appelé :
  - à la fin du bloc  si l'instance a été déclarée sur la pile (allocation statique)
  - **lors d'un appel explicite de l'opérateur delete si l'instance a été déclaré dynamiquement :**



# Appel de destructeurs

- Le destructeur est appelé :
  - à la fin du bloc  si l'instance a été déclarée sur la pile (allocation statique)
  - **lors d'un appel explicite de l'opérateur delete si l'instance a été déclaré dynamiquement :**




**Comme pour new on voit ici l'avantage par rapport au système de malloc et free. malloc et free allouent de la mémoire sans gérer les constructeurs/destructeurs, alors que new et delete les gère !**

# Destructeur par défaut

- Si aucun de destructeur n'est défini, un destructeur par défaut est appelé

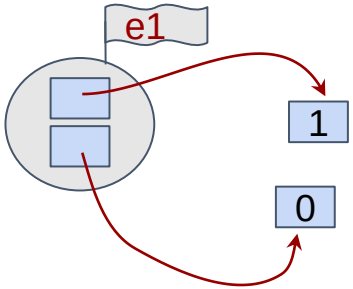
**Mais même problème que précedemment avec tout ce qui est zone de mémoire dynamique !!!**

# Destructeur par défaut

- Si aucun de destructeur n'est défini, un destructeur par défaut est appelé

```
Class Etudiant {
private:
 double* exam;
 double* devoir;
public:
 Etudiant() : devoir(new double[1]), exam(new double[0]){}
};
```

```
int main() {
  ...
  Etudiant e1;
  ...
  return 0;
}
```



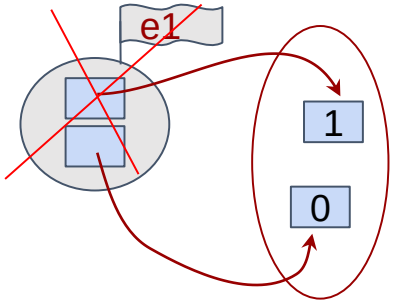
# Destructeur par défaut

- Si aucun de destructeur n'est défini, un destructeur par défaut est appelé

```
Class Etudiant {
private:
 double* exam;
 double* devoir;
public:
 Etudiant() : devoir(new double[1]), exam(new double[0]){}
};
```

```
int main() {
  ...
  Etudiant e1;
  ...
  return 0;
}
```

Appel du destructeur




ces zones mémoires existent toujours

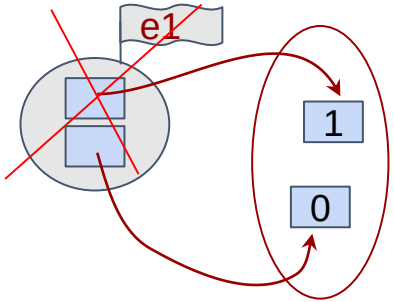
# Destructeur par défaut

- Si aucun de destructeur n'est défini, un destructeur par défaut est appelé

```
Class Etudiant {
private:
 double* exam;
 double* devoir;
public:
 Etudiant() : devoir(new double(1)), exam(new double(0)){}
};
```

```
int main() {
  ...
  Etudiant e1;
  ...
  return 0;
}
```

Appel du destructeur




ces zones mémoires existent toujours → **Il faut définir un destructeur !**

# Destructeur

- Un destructeur est une méthode particulière :
  - dont son nom est : `~NomClasse(){ … }`
  - sans aucun argument
  - sans type de retour (même pas void)

```
class MyClass{
private:
 ...
public:
 MyClass(); // constructeur par défaut
 ~MyClass(); // destructeur
};
```

- **Une classe contient un et un seul destructeur** (contrairement aux constructeurs)

# Déclaration d'un destructeur

- Syntaxe :

**MyClass.h**

```
#ifndef _Etudiant_H
#define _Etudiant_H

class MyClass{
private:
 ...
public:
 MyClass(); // constructeur par défaut
 ~Myclass(); // destructeur
 ...
};

#endif
```

**MyClass.cpp**

```
#include "Etudiant.h"

MyClass::~MyClass() {
...
      }
...
```

# Déclaration d'un destructeur

- Exemple :

**Etudiant.h**

```
#ifndef _Etudiant_H
#define _Etudiant_H

Class Etudiant {
private:
 double* exam;
 double* devoir;
public:
 Etudiant();
 ~Etudiant();

};

#endif
```

**main.cpp**

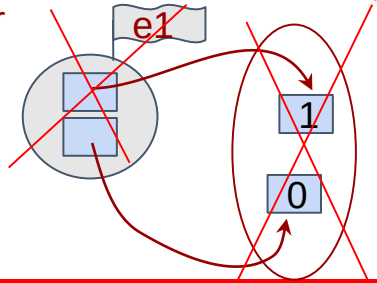
```
int main() {
  ...
  Etudiant
e1;
 ...
  return 0;
}
```

Appel du destructeur

**Etudiant.cpp**

```
#include "Etudiant.h"

Etudiant::Etudiant() : devoir(new double(1)), exam(new double(0))
{}
Etudiant::~Etudiant(){
  delete exam;
  delete devoir;
}
```

...



# Destructeur

- Règle :

Un destructeur DOIT être défini lorsqu'une classe contient des attributs alloués dynamiquement (pointeurs)

# Attributs et méthodes statiques

# Membres statiques

- Les membres (attributs ou méthodes) statiques sont attachés à la classe et non aux objets/instances créés à partir de la classe
  ⇒ Ils existent même si aucune instance n'a été créée
- Déclaration, définition et utilisation :

**MyClass.h**

```
// Déclaration
Class Myclass {
...
static type var = ...;
...
static type_retour mafonction(type1 arg1, …) ;
};
```

**main.cpp**

```
// Utilisation
int main() {
  ...
MyClass::var = val_init;
  ...

MyClass::mafonction(arg1,...);
  return 0;
}
```

**MyClass.cpp**

```
// Définition
#include "MyClass.h"

type MyClass::var = val_init;
...
type_retour MyClass::mafonction(type1
arg1, …){
...
}
...
```

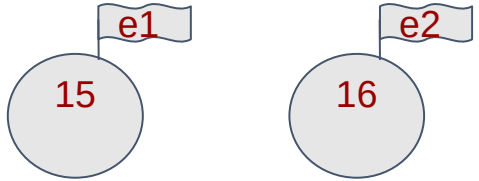
Aucun objet n'a été créé

sans **static**

# Membres statiques

- Chaque objet d'une classe a sa propre copie des attributs de la classe

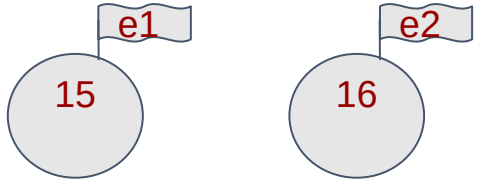
```
Etudiant e1;
Etudiant e2;
e1.setDevoir(15);
e2.setDevoir(16);
```



# Membres statiques

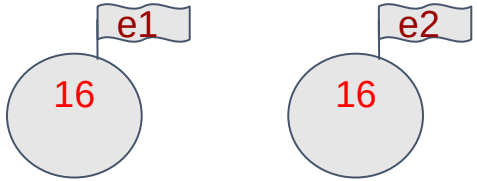
- Chaque objet d'une classe a sa propre copie des attributs de la classe

```
Etudiant e1;
Etudiant e2;
e1.setDevoir(15);
e2.setDevoir(16);
```




- Et si on déclare, dans la classe, la variable devoir comme suit ?

```
static double devoir=16;
```




la valeur de l'attribut devoir est partagé par toutes les instances de la classe Etudiant

# Utilité des membres statiques

- Retourner la date actuelle en C++ (**la méthode currentDate() est statique**) :

```
QDate::currentDate();
```

- Compteur des objets créés/détruits

- Etc…

# Exercice récapitulatif

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

# Exercice récapitulatif

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

Sortie standard

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

# Exercice récapitulatif

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

Sortie standard

```
0
```

# Exercice récapitulatif

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

Sortie standard

```
0
```

# Exercice récapitulatif

ISEN ALL IS DIGITAL! OUEST | yncréa

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

Sortie standard

```
0
2
```

# Exercice récapitulatif

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

Sortie standard

```
0
2
```

# Exercice récapitulatif

ISEN ALL IS DIGITAL! OUEST | yncréa

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

Sortie standard

```
0
2
1
```

# Exercice récapitulatif

ISEN ALL IS DIGITAL! OUEST | yncréa

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

Sortie standard

```
0
2
1
```

# Exercice récapitulatif

ISEN ALL IS DIGITAL! OUEST | yncréa

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

Sortie standard

```
0
2
1
1
```

# Exercice récapitulatif

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

Quelle est la valeur de **nbEtudiants** à la sortie du programme ?

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

# Exercice récapitulatif

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

Quelle est la valeur de **nbEtudiants** à la sortie du programme ?

**nbEtudiants = -1**

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

# Exercice récapitulatif

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

Comment procéder pour que la valeur de **nbEtudiants** à la sortie du programme soit **0** ?

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;


   return 0;
}
```

**Etudiant.cpp**

```
#include "Etudiant.h"
void Etudiant::setExam(double e) {
   exam=e;
}
void Etudiant::setDevoir(double d) {
   devoir=d;
}


int Etudiant::nbEtudiants = 0;


int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1);
   setExam(0);
   nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
```

# Exercice récapitulatif

**Etudiant.h**

```
class Etudiant {
private:
   double devoir;
   double exam;
   static int nbEtudiants;
public:
   Etudiant();
   ~Etudiant();
   void setDevoir(double d);
   void setExam(double e);
   static int getNbEtudiants();
};
```

Comment procéder pour que la valeur de **nbEtudiants** à la sortie du programme soit **0** ?

**nbEtudiants = 0**

**main.cpp**

```
#include <iostream>
#include "Etudiant.h"
using namespace std;
int main() {
   cout << Etudiant::getNbEtudiants()
<< endl;
   Etudiant e1;
   Etudiant* e2 = new Etudiant();
   cout << Etudiant::getNbEtudiants()
<< endl;
   delete e2;
   cout << Etudiant::getNbEtudiants()
<< endl;
    Etudiant e3(e1);
    cout <<
Etudiant::getNbEtudiants() << endl;

   return 0;
}
```

**Etudiant.cpp**

```
...
int Etudiant::nbEtudiants = 0;
int Etudiant::getNbEtudiants() {
   return nbEtudiants;
}
Etudiant::Etudiant() {
   setDevoir(1); setExam(0); nbEtudiants++;
}
Etudiant::~Etudiant() {
   nbEtudiants--;
}
Etudiant::Etudiant(Etudiant const&
autre) {
   setDevoir(autre.devoir);
   setExam(autre.exam);
   nbEtudiants++;
}
```

# FIN !