

Support de cours

Cours:

Introduction à la programmation orientée objet (en C++)

Vidéo:

W17-05-copieprofonde-CPP-pt2

Concepts (extraits des sous-titres générés automatiquement) :

Constructeur de copie. Valeur du pointeur. Copie des différents composants. Cœur de l'instance courante. Nouvelle zone mémoire. Égal m1. Copie de surface. Opérateur d'affectation. Copie polymorphique. Telle copie polymorphique. Copie de l'instance courante. Méthode copie. Copie du contenu. Spécificités de l'instance. Niveau de la classe.



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en C++).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>

Etude de cas : copie profonde

(Partie 2)

Introduction à la programmation orientée objet (en C++)

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s





sinon en effet c'est un appel au constructeur par défaut qui aurait lieu ici si on avait pas écrit cette ligne là

notes

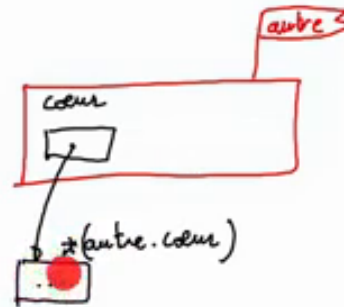
résumé

0m 1s



Constructeur de copie

```
Montre(const Montre& autre)
// Ne pas oublier d'appeler le constructeur DE COPIE de la super-classe
: Produit(autre),
  coeur(???)
// ...??
{
  // ...??
}
```



donc on aurait à ce moment là une construction par défaut de la partie Produit d'une montre, c'est à dire, mise à zéro par exemple de son prix ce qui n'est pas forcément toujours souhaitable. Donc il faut absolument toujours commencer par un appel au constructeur de copie des super-classes. Ensuite la question c'est, comment continuer, quand écrire la copie des différents composants ? Si l'on écrit simplement comme ceci et de façon similaire pour les accessoires alors à ce moment là on va avoir une copie de surface, on aura copié la valeur du pointeur (autre.coeur) dans le pointeur "coeur" de l'instance qu'on est en train de créer c'est donc justement exactement ce que l'on ne veut pas donc ce n'est pas la bonne façon de faire. Ce que l'on peut alors imaginer, c'est effectivement copier l'objet pointé par le coeur de l'autre, donc faire une copie de *(autre.coeur), en effet l'accès au contenu pointé par le coeur de l'autre

notes

résumé

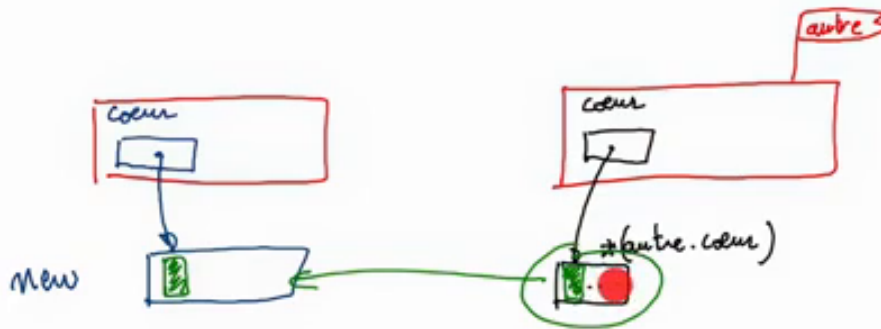
0m 12s



Constructeur de copie

```
Montre(const Montre& autre)
// Ne pas oublier d'appeler le constructeur DE COPIE de la super-classe
: Produit(autre),
  coeur(???)
// ...??
{
  // ...??
}

coeur(unique_ptr<Mecanisme>(new Mecanisme(*(autre.coeur))));
unique_ptr<Mecanisme>
```



c'est bien le contenu de cette cellule ici c'est bien ça que l'on voulait copier. Copier dans le cœur de l'instance courante et pour ceci donc, allouer une nouvelle zone mémoire dont on récupérerait l'adresse et cette nouvelle zone là étant la copie du contenu de (autre.cœur). Donc ça s'écrirait comme ceci : coeur(new ??? (*(autre.cœur)) - mais "new" quoi ? quel objet devrait être copié en tant que cœur de l'autre ? Puisque je vous rappelle qu'un cœur c'est un pointeur sur un mécanisme et donc ici, il faudrait bien sûr convertir notre pointeur à la C on met unique_ptr mais on ne sait toujours pas en tant que quoi il faut copier cet autre cœur parce que cet autre cœur pourrait très bien être un mécanisme analogique ou un mécanisme digital ou un mécanisme double et donc, en tant que quoi copier cet autre cœur ? Ce que l'on pourrait imaginer c'est de le copier en tant que Mecanisme mais à ce moment là ça veut dire que l'on recopie que la partie Mecanisme du cœur de l'autre et l'on risque de perdre toutes les spécificités

notes

résumé

1m 13s



Constructeur de copie

```
Montre(const Montre& autre)
// Ne pas oublier d'appeler le constructeur DE COPIE de la super-classe
: Produit(autre),
  coeur(???)
// ...??
{
  // ...??
}
```

Mais... comment copier chaque élément en tant que tel ?
(et non pas comme une instance des super-classes *Mecanisme* et *Accessoire*)

☞ **copie polymorphique**

de l'instance réellement pointée donc les spécificités analogiques, si c'est un cœur qui est un mécanisme analogique les spécificités digitales et cetera, on perdrait donc, les spécificités. Donc ici, on ne peut pas faire une copie comme cela en tant que *Mecanisme*. La question de fond, c'est :

notes

résumé

2m 37s



```
Montre(const Montre& autre)
// Ne pas oublier d'appeler le constructeur DE COPIE de la super-classe
: Produit(autre),
  coeur(autre.coeur->copie())
{
  for (auto const& p_acc : autre.accessoires) {
    accessoires.push_back(p_acc->copie());
  }
}
```

comment copier chaque élément en tant que tel ? C'est à dire ne pas les copier en tant qu'instance uniquement de super-classes de Mekanisme mais de les copier en tant que sous-classes pour ceux qu'elles sont elles-mêmes. Et la question se pose de la même façon bien sûr pour tous les accessoires, comment copier chacun des accessoires ? Le bracelet en tant que bracelet, les fermoirs en tant que fermoirs, les boîtiers en tant que boîtiers et cetera. La question est donc, comment fait on une copie qui garde la nature propre des instances ? c'est à dire, comment fait on une copie polymorphique ? La solution consiste, comme à chaque fois qu'on a une tâche particulière à faire à définir une méthode qui fait cette tâche et donc ici, à définir une méthode copie qui est capable de faire une copie polymorphique, ici une copie polymorphique de Mekanisme et ici une copie polymorphique d'accessoires.

notes

résumé

2m 58s



Copie polymorphique

```
vector<unique_ptr<Accessoire>> accessoires;  
// ...  
accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphique d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};

//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphique de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```

Il faudra alors pour le coup, copier évidemment chaque accessoire en tant que tel et donc parcourir les accessoires de l'autre montre un par un pour pouvoir chacun, le copier de façon polymorphique. Voyons donc comment s'écrit une telle copie polymorphique et prenons pour cela l'exemple des accessoires.

notes

résumé

3m 49s



Copie polymorphique

```
vector<unique_ptr<Accessoire>> accessoires;
// ...
accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphique d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};
//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphique de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```

Nous avons donc, un tableau dynamique de pointeurs sur des accessoires dont on veut copier chacun des éléments et typiquement donc on le copierait comme ceci en faisant un appel à une copie polymorphique

notes

résumé

4m 6s



Copie polymorphique

```
vector<unique_ptr<Accessoire>> accessoires;  
// ...  
accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphique d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};

//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphique de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```

d'un pointeur donc sur des accessoires. Donc la copie polymorphe va être défini au niveau de la classe la plus générale pour laquelle on veut faire une telle copie polymorphe à ce niveau là on va dire qu'il existe une méthode copie, la méthode copie en tant que telle, ne modifie pas les instances elle les copie ailleurs et on ne sait pas la définir à un niveau aussi général typiquement pour une classe abstraite donc ici on va dire

notes

résumé

4m 20s



Copie polymorphique

```
vector<unique_ptr<Accessoire>> accessoires;
// ...
accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphique d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};
//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphique de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```

que c'est une méthode virtuelle pure. La valeur de retour d'une telle méthode c'est ce que l'on doit mettre ici dans cette collection et donc ça sera un `unique_ptr`

notes

résumé

4m 44s



Copie polymorphique

```
vector<unique_ptr<Accessoire>> accessoires;
// ...
accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphique d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};
//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphique de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```

donc ici on va dire que c'est un `*unique_ptr`. Et donc dans les sous-classes par exemple ici dans la sous-classe `Bracelet` ce que l'on va faire c'est que l'on va redéfinir la copie qui retourne un `unique_ptr` en disant que l'on va retourner en tant que `unique_ptr` sur un accessoire, donc ça, ça reste un pointeur sur un accessoire mais ce pointeur est spécifiquement un pointeur sur un nouveau bracelet en mémoire qui est une copie de l'instance courante. Alors, je sais que cette syntaxe est extrêmement difficile à comprendre la première fois qu'on la rencontre,

notes

résumé

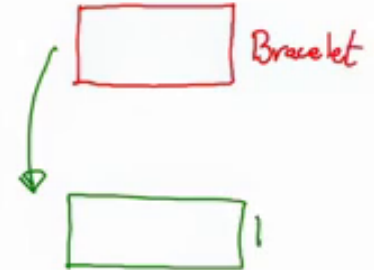
4m 56s



Copie polymorphique

```
vector<unique_ptr<Accessoire>> accessoires;
// ...
accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphique d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};
//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphique de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```



détaillons la donc un petit peu. Ce que l'on veut faire avec une telle méthode copie c'est se copier en tant que tel

notes

résumé

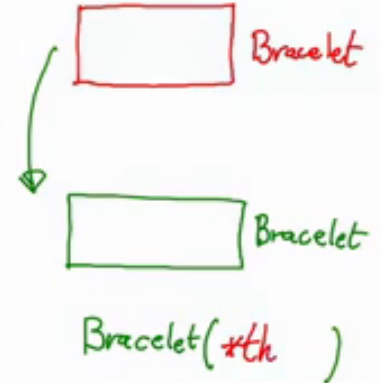
5m 30s



Copie polymorphique

```
vector<unique_ptr<Accessoire>> accessoires;
// ...
accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphique d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};
//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphique de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```



donc se copier en tant que Bracelet si l'on est un Bracelet, pour ceci on va appeler le constructeur de copie de la classe bracelet pour copier l'instance courante, l'instance courante c'est bien

notes

résumé

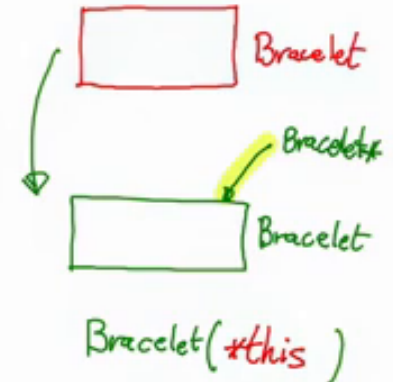
5m 41s



Copie polymorphique

```
vector<unique_ptr<Accessoire>> accessoires;
// ...
accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphique d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};
//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphique de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```



le contenu du pointeur this donc (*this) c'est l'instance courante c'est le contenu de ce qui est pointé par this this pointant l'instance courante ceci est donc bien l'instance courante que l'on copie dans un nouveau Bracelet. Donc ça nous donne cette partie là de cette ligne et puis on va faire ceci en créant une nouvelle zone mémoire et récupérer un pointeur sur cette zone mémoire ce qui est fait par le new, donc ici new Bracelet ça va nous créer un nouveau bracelet dont on récupère l'adresse et ce nouveau bracelet, c'est une copie de l'instance courante. Et enfin, ce pointeur à la 'c'

notes

résumé

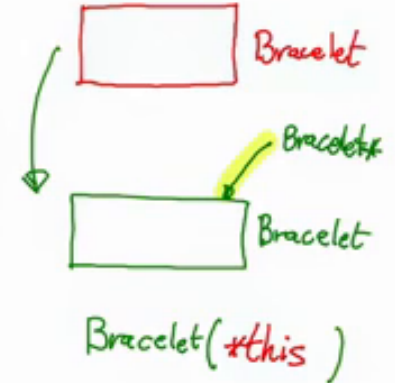
5m 57s



Copie polymorphique

```
vector<unique_ptr<Accessoire>> accessoires;
// ...
accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphique d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};
//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphique de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```



résultat du new de ce Bracelet qui copie l'instance courante, on va le transformer en un unique_ptr on va le voir, on va regarder ce pointeur

notes

résumé

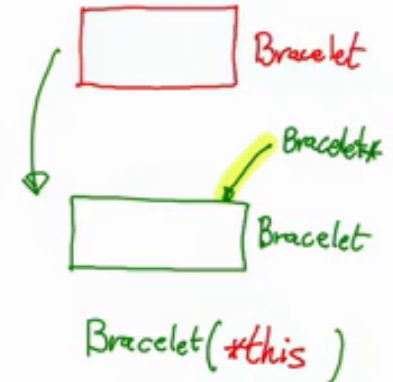
6m 45s



Copie polymorphique

```
vector<unique_ptr<Accessoire>> accessoires;
// ...
accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphique d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};
//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphique de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```



comme un unique_ptr sur un accessoire. Voilà donc la formulation utilisée très souvent pour faire des copies polymorphique on return un pointeur sur la classe abstraite, la super-classe au niveau de laquelle on veut faire la copie

notes

résumé

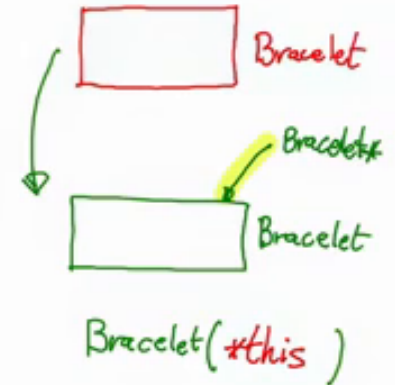
6m 57s



Copie polymorphique

```
vector<unique_ptr<Accessoire>> accessoires;
// ...
accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
    // ...
    // copie polymorphique d'Accessoire
    virtual unique_ptr<Accessoire> copie() const = 0;
    // ...
};
//-----
class Bracelet : public Accessoire {
public:
    // ...
    // copie polymorphique de Bracelet
    virtual unique_ptr<Accessoire> copie() const override
    { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```



d'une nouvelle copie ici, cette fois-ci attention, spécifique de la classe courante, donc, ceci c'est bien dans la classe Bracelet on met bracelet,

notes

résumé

7m 13s





dans la classe Fermoir on mettra fermoir copie de l'instance courante.

notes

résumé

7m 20s



```
class Montre : public Produit {
public:
    // ...
    Montre& operator=(Montre source) // Notez le passage par VALEUR
    {
        swap(coeur      , source.coeur      );
        swap(accessoires, source.accessoires);
        return *this;
    }
};
```

Voilà donc pour la copie des montres. Quand est-il si on veut aussi faire l'opérateur d'affectation ? C'est à dire par exemple, faire m2 égal m1. Pour ceci nous allons surcharger l'opérateur d'affectation l'opérateur égal, de la façon suivante, dans la classe Montre, suit le schéma que l'on a présenté dans les vidéos sur la surcharge des opérateurs

notes

résumé

7m 28s



m2 =

```
class Montre : public Produit {
public:
    // ...
    Montre& operator=(Montre source) // Notez le passage par VALEUR
    {
        swap(coeur      , source.coeur      );
        swap(accessoires, source.accessoires);
        return *this;
    }
};
```

et donc, on a ici le prototype typique de l'opérateur égal où l'on utilise ici, un passage par valeur

notes

résumé

7m 49s



$m2 = m1;$ $m2.operator=(m1)$

```
class Montre : public Produit {
public:
    // ...
    Montre& operator=(Montre source) // Notez le passage par VALEUR
    {
        swap(coeur      , source.coeur      );
        swap(accessoires, source.accessoires);
        return *this;
    }
};
```

ce qui fait qu'effectivement lorsqu'on va écrire $m2=m1$ c'est à dire,
 $m2.operator=(m1)$

notes

résumé

8m 0s



$m2 = m1;$ $m2.operator=(m1)$

```
class Montre : public Produit {  
public:  
    // ...  
    Montre& operator=(Montre source) // Notez le passage par VALEUR  
    {  
        swap(coeur      , source.coeur      );  
        swap(accessoires, source.accessoires);  
        return *this;  
    }  
};
```

on aura ici copie de m1 dans cette nouvelle source. Donc, on utilise ici le constructeur de copie

notes

résumé

8m 7s



$m2 = m1;$ $m2.operator=(m1)$

```
class Montre : public Produit {  
public:  
    // ...  
    Montre& operator=(Montre source) // Notez le passage par VALEUR  
    {  
        swap(coeur, source.coeur);  
        swap(accessoires, source.accessoires);  
        return *this;  
    }  
};
```

et c'est ce constructeur de copie que l'on vient justement de définir qui va faire tout le travail de la copie profonde ensuite on a plus qu'à échanger l'instance nouvellement copiée ex m1 devenu source dans l'instance courante et pour ceci, donc par exemple on peut, en utilisant la méthode swap fournie dans la bibliothèque standard utility, échanger le cœur de m2 de l'instance courante avec le cœur de la source nouvellement créée par copie et donc échanger le tableau des pointeurs de l'instance courante avec le tableau des pointeurs vers des nouvelles composantes créée par copie profonde de cette source donc justement, nouvellement créée

notes

résumé

8m 17s



$m2 = m1;$ $m2.operator=(m1)$

```
class Montre : public Produit {
public:
    // ...
    Montre& operator=(Montre source) // Notez le passage par VALEUR
    {
        swap(coeur, source.coeur);
        swap(accessoires, source.accessoires);
        return *this;
    }
};
```

$m3 = m2 = m1;$

et on termine avec le schéma classique qui retourne une référence sur l'instance courante pour être conforme à ce prototype qui permet des écritures du genres $m3=m2=m1$. Voilà donc pour l'opérateur d'affectation

notes

résumé

8m 59s



Test : un exemple de main()

```
// ... (le reste du main() comme avant)
// Nous faisons une copie de la montre m
Montre m2(m);
cout << "Montre m2 :" << endl;
cout << m2 << endl;

// Et testons l'opérateur d'affectation
Montre m3(new MecanismeAnalogique(87.00, 20140415));
cout << "Montre m3 (1) :" << endl;
cout << m3 << endl;

m3 = m2; m2.mettre_a_1_heure("10:10");
cout << "Montre m3 (2) :" << endl;
cout << m3 << endl;
// ...
```

Le code complet à ce stade (367 lignes) peut être téléchargé sur le site du cours :

<https://d396qusza40orc.cloudfront.net/cpp-fr/complements/montres.cc>

et bien sûr tout ceci doit être testé,

notes

résumé

9m 13s





ce que nous ferions typiquement complétant notre ancien `main` dans lequel nous avons défini une montre par un appel ici au constructeur de copie qui copie `m` dans `m2`, puis l'on vérifie que la copie s'est faite correctement en affichant ici `m2` et puis on testerait l'opérateur égal. d'abord on crée ici une nouvelle montre `m3` avec peu importe ce qu'il y a dedans et puis on l'affiche, on fait ensuite une affectation de `m2` à `m3` et on vérifie que effectivement cette affectation a fonctionné correctement. Pour vérifier en plus que nous avons bien fait une copie profonde dans cette copie ici, alors on ajouterait une méthode que nous n'avons pas détaillée dans cette vidéo mais qui ne présente aucune difficulté une méthode `mettre à l'heure` qui mettrait la montre `m2` à l'heure qui modifierait l'heure de la même montre : `m2` et on vérifierait donc que dans cette affectation c'est bien `m3` qui avait l'ancienne heure de `m2` et qui n'est pas affectée par cette mise à l'heure de `m2` si on avait qu'une copie de surface alors ici `m2.mettre_a_l_heure` remettrait aussi à l'heure `m3`. Donc cette ligne ici nous permet effectivement de vérifier que nous avons une copie profonde et non pas une copie de surface de `m2` dans `m3`. Tout ce code, tout le code complet est à nouveau fourni sur le site du cours. Voilà donc tout ceci conclut cette étude de cas et donc termine tout ce cours. et donc termine tout ce cours.

notes

résumé

9m 15s

