

Chapitre 7

Classes : deuxième partie

Auteur : Marco Lavoie
Adaptation : Sébastien Bois



Langage C++
25908 ORD



Objectifs

- Créer et détruire des objets dynamiquement
- Manipuler des objets **const**
- Comprendre l'utilité de l'attribut **friend**
 - Fonctions **friend**
 - Classes **friend**
- Comprendre l'utilité du pointeur **this**



Aperçu

- Objets `const` et fonctions membres `const`
- Objets comme attributs membres
- Fonctions `friend` et classes `friend`
- Utilisation du pointeur `this`
- Allocation dynamique de mémoire avec les opérateurs `new` et `delete`
- Membres de classes `static`
- La macro `assert`



Introduction

- Le langage C++ est complexe car il offre une flexibilité accrue en rapport aux autres langages
 - Allocation statique et dynamique d'objets
 - Objets constants
 - Fonctions et classes amies
 - Pour assurer la surcharge d'opérateurs
- Ce chapitre présente ces particularités du C++ en préparation des chapitres suivants



Objets et fonctions membres **const**

- Le mot-clé **const** permet de définir une variable en lecture seulement

```
const int taille = 20;
```

- Similairement, on peut instancier une classe sous forme d'objet en lecture seulement

```
const Temps heureSouper( 17, 0, 0 );
```

- La variable `heureSouper` est une constante fixée à 17:00:00 et ne pouvant pas être modifiée (i.e. en lecture seulement)



Rappel : variable **const** et paramètre référence

- Que se passe-t-il dans l'exemple suivant ?

```
void fonc( int & x ) {  
    x = 1;  
}  
  
int main() {  
    const int a = 0;  
  
    fonc( a );  
    std::cout << a;  
  
    return 0;  
}
```

- L'invocation **fonc(a)** fait en sorte que la fonction tente de modifier la valeur de **a** via le paramètre référence **x**
- Conséquence : *erreur de compilation*
 - Le compilateur ne permet pas l'invocation



Rappel : impact d'une fonction membre

- Une fonction membre peut modifier l'objet pour lequel elle s'exécute

```
class Temps {  
    int heure, minute, seconde;  
public:  
    Temps( int = 0, int = 0, int = 0 );  
    void ajusterTemps( int, int, int );  
};
```

```
void Temps::ajusterTemps( int h, int m, int s ) {  
    heure    = h;  
    minute   = m;  
    seconde  = s;  
}
```

```
int main() {  
    Temps t( 0, 0, 0 );  
    t.ajusterTemps( 17, 4, 39 );  
}
```

Si la variable **t** est déclarée constante alors ça ne compilerait pas

```
const Temps t( 0, 0, 0 );
```

Les attributs de la variable **t** sont modifiés par l'invocation



Rappel : impact d'une fonction membre (suite)

- Et qu'en est-il d'une fonction membre ne modifiant pas l'objet ?

```
class Temps {  
    int heure, minute, seconde;  
public:  
    Temps( int = 0, int = 0, int = 0 );  
    void ajusterTemps( int, int, int );  
    bool validerTemps();  
};  
  
bool Temps::validerTemps() {  
    return ( heure    >= 0 && heure    <= 23 ) &&  
           ( minute   >= 0 && minute   <= 59 ) &&  
           ( seconde  >= 0 && seconde  <= 59 );  
}  
  
int main() {  
    const Temps t( 0, 0, 0 );  
    bool res = t.validerTemps(); x  
}
```

- Ça ne compile pas non plus
 - Le compilateur ne peut pas efficacement déterminer si la fonction `validerTemps()` va occasionner la modification de l'objet
- Est-ce à dire qu'une fonction membre ne peut traiter un objet constant ?



Rappel : impact d'une fonction membre (suite)

- Non, mais il faut indiquer au compilateur que la fonction membre ne modifie pas l'objet sur lequel elle s'exécute

```
class Temps {  
    int heure, minute, seconde;  
public:  
    Temps( int = 0, int = 0, int = 0 );  
    void ajusterTemps( int, int, int );  
    bool validerTemps() const;  
};  
  
bool Temps::validerTemps() const {  
    return ( heure >= 0 && heure <= 23 ) &&  
           ( minute >= 0 && minute <= 59 ) &&  
           ( seconde >= 0 && seconde <= 59 );  
}  
  
int main() {  
    const Temps t( 0, 0, 0 );  
    bool res = t.validerTemps();  
}
```

*En indiquant que la fonction membre est **const**, on certifie au compilateur que celle-ci ne modifie pas les attributs de l'objet*



Fonctions membres **const**

- Elles peuvent être invoquées par des objets **const** ou non

```
class Temps {  
    int heure, minute, seconde;  
public:  
    Temps( int = 0, int = 0, int = 0 );  
    void ajusterTemps( int, int, int );  
    bool validerTemps() const;  
};
```

```
int main() {  
    const Temps tc( 0, 0, 0 );  
    Temps tv( 0, 0, 0 );  
  
    tc.validerTemps();  
  
    tv.validerTemps();  
    tv.ajusterTemps( 17, 4, 39 );  
}
```

← Notez l'absence de
tc.ajusterTemps(17, 4, 39);
car elle ne peut être invoquée sur tc



Fonctions membres **const** (suite)

- Et si une fonction membre **const** tente de modifier les attributs de l'objet ?

```
class Temps {  
    int heure, minute, seconde;  
public:  
    Temps( int = 0, int = 0, int = 0 );  
    void ajusterTemps( int, int, int ) const;  
};
```

```
void Temps::ajusterTemps( int h, int m, int s ) const {  
    heure    = h;  ✗  
    minute   = m;  ✗  
    seconde  = s;  ✗  
}
```

– Ça ne compile pas car la fonction membre n'est pas autorisée à modifier les attributs de l'objet



Fonctions membres **const** (suite)

- Et qu'en est-il des constructeurs ?

- On ne déclare jamais les constructeurs avec le qualificatif **const**, car ceux-ci peuvent modifier l'objet
- En C++, on ne peut pas définir un constructeur **const**

```
class Temps {  
    int heure, minute, seconde;  
public:  
    Temps( int = 0, int = 0, int = 0 );  
};  
  
Temps::Temps( int h, int m, int s ) {  
    heure    = h;  
    minute   = m;  
    seconde  = s;  
}  
  
int main() {  
    const Temps tc( 17, 4, 39 ); ✓  
    return 0;  
}
```



Attributs membres **const**

- Un attribut membre peut être **const**
 - Mais le constructeur ne peut pas initialiser cet attribut car il est **const**
 - Un attribut ne peut pas non être initialisé dans la déclaration de la classe
- Alors comment l'initialiser ?

```
class Increment {  
public:  
    Increment( int = 0, int = 1 );  
    void Incrementer();  
    int getCompteur();  
private:  
    int compteur;  
    const int increment, = 1;  
};  
  
Increment::Increment( int c, int i ) {  
    compteur = c;  
    increment = i;  
}  
  
void Increment::Incrementer() {  
    compteur += increment;  
}  
  
int Increment::getCompteur() {  
    return compteur;  
}
```



Attributs membres **const** (suite)

- Attributs membres **const** initialisés dans le constructeur via un *initialiseur*

- Énumération des attributs à initialiser avec sa valeur initiale entre parenthèses, séparés par des virgules
- Les initialiseurs peuvent aussi initialiser les attributs non constants

```
class Increment {  
public:  
    Increment( int = 0, int = 1 );  
private:  
    int      compteur;  
    const int increment;  
};  
  
Increment::Increment( int c, int i )  
    : increment( i ) {  
    compteur = c;  
}
```


```
Increment::Increment( int c, int i )  
    : increment( i ), compteur( c ) {  
}
```

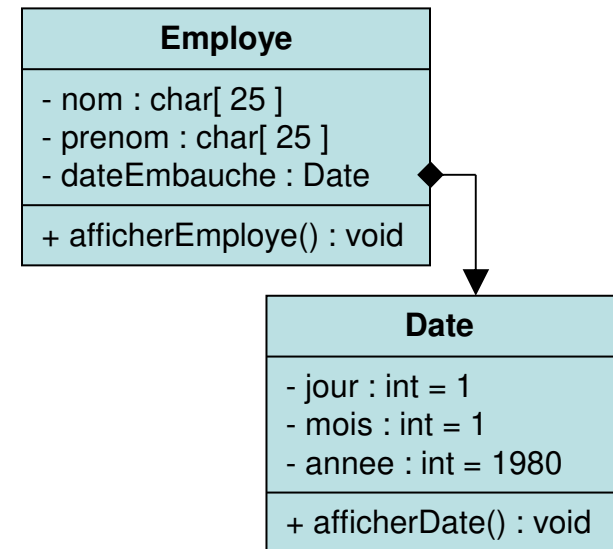
Notez que le bloc de code du constructeur est vide puisqu'il n'y a rien d'autre à y mettre

Objets membres

- Un attribut membre peut être un objet (i.e. instance) d'une autre classe

```
class Date {  
public:  
    Date( int = 1, int = 1, int = 1980 );  
    void afficherDate();  
private:  
    int jour, mois, annee;  
};  
  
class Employe {  
public:  
    Employe( char [], char [], int, int, int );  
private:  
    char nom[ 25 ], prenom[ 25 ];  
    Date dateEmbauche;  
    void afficherEmploye();  
};
```

– En UML, la flèche  représente cette relation





Objets membres : initialiseurs

- Comment initialiser un objet membre ?
 - Avec des initialiseurs exploitant un constructeur de l'objet membre

```
Employe::Employe( char nm[], char pnm[], int j, int m, int a )  
    : dateEmbauche( j, m, a ) {  
    strncpy( nom, nm, 25 );  
    strncpy( prenom, pnm, 25 );  
}
```

- C'est la seule façon d'initialiser `dateEmbauche` car ses attributs `jour`, `mois` et `annee` étant privés, la classe `Employe` n'y a pas directement accès



Fonctions amies

- Une fonction n'a pas directement accès aux membres privés d'une classe
- Il y a parfois des circonstances dans lesquelles on doit autoriser une fonction à accéder aux membres privés d'une classe
 - Nous verrons ces circonstances plus tard

```
class Date {  
public:  
    Date( int, int, int );  
private:  
    int jour, mois, annee;  
};
```

```
void ajusterDate( Date &d, int j,  
                 int m, int a ) {  
  
    d.jour = j; ×  
    d.mois = m; ×  
    d.annee = a; ×  
};
```

*Erreur de compilation car
ajusterDate n'est pas
une fonction membre de
la classe Date*



Fonctions amies (suite)

- Une classe peut cependant autoriser des fonctions à accéder à ses membres privés
 - Ces fonctions sont dites *amies* (**friend**) de la classe
 - Il suffit de spécifier le prototype de la fonction amie dans la classe, précédée du mot-clé **friend**

```
class Date {  
    friend void ajusterDate( Date &, int, int, int );  
public:  
    Date( int, int, int );  
private:  
    int jour, mois, annee;  
};
```

```
void ajusterDate( Date &d, int j,  
                 int m, int a ) {  
  
    d.jour  = j; ✓  
    d.mois  = m; ✓  
    d.annee = a; ✓  
  
};
```

Maintenant ça
compile



Classes amies

- Comme une fonction, une classe peut aussi être amie d'une autre classe

- Dans l'exemple ci-contre, la classe `Employe` a accès aux attributs privés de la classe `Date`
- L'amitié n'est pas réciproque
 - `Date` n'a pas accès aux membres privés de `Employe`

```
class Date {  
    friend class Employe;  
public:  
    Date( int = 1, int = 1, int = 1980 );  
    void afficherDate();  
private:  
    int jour, mois, annee;  
};  
  
class Employe {  
public:  
    Employe( char [], char [], int, int, int );  
private:  
    char nom[ 25 ], prenom[ 25 ];  
    Date dateEmbauche;  
    void afficherEmploye();  
};
```



Exercice 7.1

20

- Considérez la déclaration de classe suivante :

```
const int MAX = 100;
class Donnees {
public:
    Donnees();                // constructeur par défaut (aucune données)
    Donnees( double *, int ); // constructeur paramétré
    double moyenne();         // calcule la moyenne des données

private:
    double valeurs[ MAX ];    // stockage de données
    int     nbValeurs;         // nombre de données dans valeurs[]
};
```

- Complétez cette classe de sorte qu'elle fonctionne avec le programme ci-contre

```
int main() {
    const double v[] = { 3.2, 4.6,
                        2.9 };
    const Donnees data( v, 3 );
    cout << data.moyenne() << endl;
    return 0;
}
```



Pointeur implicite `this`

- Chaque instance de classe (i.e. objet) a accès à sa propre adresse mémoire via le pointeur `this`
 - Chaque fonction membre d'une classe a accès implicitement au pointeur `this` de l'objet pour lequel elle s'exécute
 - Seule exception : les *fonctions membres statiques* (qu'on verra dans les prochaines acétates)
 - Principale utilisation : retourner l'objet ou son adresse

this est exploité implicitement

- Même si on ne fait pas référence à **this**, le compilateur le fait implicitement

```
class Test {  
public:  
    Test( int = 0 );  
    void afficher();  
private:  
    int x;  
};  
  
Test::Test( int a ) {  
    x = a;  
}  
  
void Test::afficher() {  
    cout << "          x = " << x  
        << "\n  this->x = " << this->x  
        << "\n(*this).x = " << (*this).x;  
}
```

```
int main() {  
    Test t( 12 );  
    t.afficher();  
  
    return 0;  
}
```

```
C:\>test.exe  
          x = 12  
  this->x = 12  
(*this).x = 12  
C:\>
```



`this` comme valeur de retour

- Une fonction membre peut retourner `this` afin de permettre le *chaînage*

```
class Test {  
public:  
    Test( int = 0 );  
    Test & incr();  
private:  
    int x;  
};
```

```
Test::Test( int a ) {  
    x = a;  
}
```

```
Test & Test::incr() {  
    x++;  
    return *this;  
}
```

Retourne une
référence à l'objet

```
int main() {  
    Test t( 0 );  
  
    t.incr();    // t.x = 1  
    t.incr().incr().incr(); // t.x = 4  
  
    return 0;  
}
```

Ça fonctionne car `t.incr()`
retourne l'objet, pour lequel
on invoque à nouveau la
fonction membre

– Nous verrons plus tard
l'utilité du *chaînage*



Allocation dynamique

- Opérateurs d'allocation dynamique
 - **new** : permet de réserver un espace mémoire durant l'exécution du programme
 - **delete** : permet de libérer un espace mémoire (durant l'exécution) obtenu précédemment via un new
- Distinction importante
 - Allocation statique : le compilateur gère le bloc mémoire
 - Allocation dynamique : le programmeur gère le bloc mémoire avec **new** et **delete**



Opérateur **new**

- Retourne un *pointeur* au bloc mémoire alloué
 - Exemple :

```
Date *p;  
p = new Date( 24, 2, 2010 );
```
- Notes importantes
 - Le type du pointeur doit correspondre au type spécifié après **new**
 - **new** permet d'invoquer un constructeur si le type spécifié est une classe



Allocation statique vs dynamique

- Distinction entre les deux types d'allocation
 - À l'exécution

```
{  
    Date d( 12, 11, 1998 );  
    d.afficher();  
  
    Date *p;  
    p = new Date( 24, 2, 2009 );  
    p->afficher();  
}
```



- À la fin d'exécution du bloc, l'objet `Date` créé par l'instruction `new` demeure en mémoire :

jour	: 24
mois	: 2
annee	: 2009

- Le bloc mémoire associé à l'objet statique `d` est automatiquement géré par le compilateur



Opérateur `delete`

- Libère un bloc mémoire obtenu via l'opérateur `new`

– Exemple à l'exécution :

```
{  
    Date d( 12, 11, 1998 );  
    d.afficher();  
  
    Date *p;  
    p = new Date( 24, 2, 2009 );  
    p->afficher();  
    delete p;  
}
```



- L'instruction `delete` détruit le bloc mémoire spécifié
- Notez qu'on donne à `delete` l'adresse mémoire du bloc à détruire (adresse dans le pointeur `p`)



Gestion dynamique de tableaux

- Les opérateurs `new` et `delete` permettent de gérer dynamiquement des tableaux
 - En conjonction avec l'opérateur `[]`
 - Puisque `t` est un pointeur, il faut indiquer à `delete` que celui-ci pointe à un tableau, d'où le besoin d'inclure l'opérateur `[]` après le `delete`

```
{  
    int *t = new int[ 10 ];  
  
    t[0] = 14;  
    t[1] = 17;  
    ...  
  
    delete [] t;  
}
```



Tableaux dynamiques

- Pourquoi gérer un tableau de façon dynamique plutôt que statique ?

```
int main() {  
    int n;  
  
    cout << "Nombre de valeurs ? ";  
    cin >> n;  
  
    int t[ n ]; // créer le tableau ✗  
  
    for ( int i = 0; i < n; i++ )  
        cin >> t[ i ];  
  
    ...  
}
```

← Mémoire alloué à t ici

- Le tableau t ne peut pas être de taille n car son bloc mémoire est alloué au début du bloc, mais la valeur de n est lue dans ce même bloc!

↘ Taille du tableau t seulement connue ici



Tableaux dynamiques (suite)

- Si on se limite à l'allocation statique, on doit limiter la flexibilité du programme

```
int main() {  
    const unsigned int N = 10;
```

```
    do {  
        cout << "Nombre de valeurs ? ";  
        cin >> n;  
    } while ( n > N );
```

*Le programme ne
pourra jamais traiter
plus de 10 valeurs*

```
    int t[ N ]; // créer un tableau approprié
```

```
    for ( int i = 0; i < n; i++ )  
        cin >> t[ i ];
```

```
    ...
```

```
}
```



Tableaux dynamiques (suite)

- L'allocation dynamique permet d'allouer la mémoire à t seulement lorsque la valeur de n est connue

```
int main() {  
    unsigned int n;  
  
    cout << "Nombre de valeurs ? ";  
    cin >> n;  
  
    int *t = new int[ n ];  
  
    for ( int i = 0; i < n; i++ )  
        cin >> t[ i ];  
    ...  
  
    delete [] t;  
}
```

← Aucune limite de nombre de valeurs à traiter (sauf celle imposée par la quantité de RAM dans l'ordinateur)



Dangers de l'allocation dynamique

- Lorsqu'on fait un `new`, le programmeur ne doit pas oublier d'éventuellement faire le `delete` correspondant
 - Sinon il y aura *fuite de mémoire*

```
{  
    Date d( 12, 11, 1998 );  
    d.afficher();  
  
    Date *p;  
    p = new Date( 24, 2, 2009 );  
    p->afficher();  
}
```

jour	: 24
mois	: 2
annee	: 2009



Exercice 7.2

- Considérez la classe de l'exercice 7.1 modifiée :

```
class Donnees {  
public:  
    Donnees( int = 100 );           // constructeur par défaut  
    Donnees( double *, int, int = 100 ); // Constructeur paramétré  
  
    int    ajout( double );        // ajoute une donnée à valeurs[]  
    double moyenne() const;       // calcule la moyenne des données  
private:  
    double *valeurs;              // stockage de données  
    int     nbValeurs,            // nombre de données dans valeurs[]  
           szValeurs;            // taille du tableau valeurs[]  
};
```

- Complétez cette classe de sorte qu'elle fonctionne avec le programme ci-contre

```
int main() {  
    double v[] = { 3.2, 4.6, 3.9 };  
    Donnees data( v, 3 );  
  
    data.ajout( 1.7 ).ajout( 9.4 );  
    cout << data.moyenne() << endl;  
  
    return 0;  
}
```



Membres de classe **static**

- Les membres d'une classe peuvent être déclarés **static**
 - Attention : ceci n'a rien à voir avec l'allocation statique
- Un attribut membre **static** permet à toutes les instances d'une même classe de partager le même attribut
 - Contrairement aux attributs membres conventionnels où chaque instance a sa propre copie de l'attribut



Attributs membres **static**

- Exemple

```
class Test {  
public:  
    Test( int = 0 );  
    static int nbInstances();  
private:  
    int x;  
    static int n;  
};  
  
// Initialiser le membre statique  
int Test::n = 0;  
  
Test::Test( int a ) {  
    x = a;  
    → n++; // Compter les instances  
}  
  
int Test::nbInstances() {  
    return n;  
}
```

```
int main() {  
    for ( int i = 0; i < 100; i++ ) {  
        Test t;  
    }  
  
    cout << "Nombre d'instances créés = "  
    → << Test::nbInstances() << endl;  
  
    return 0;  
}
```

- L'attribut **static n** est accessible par les instances et par le public via une fonction membre **static** (nbInstances), et ce même sans passer par une instance



Attributs membres **static** (suite)

- Attention : une fonction membre statique peut s'exécuter sans faire référence à un objet instancié de la classe

```
int main() {  
    for ( int i = 0; i < 100; i++ ) {  
        Test t;  
    }  
  
    cout << "Nombre d'instances créés = "  
        << Test::nbInstances() << endl;  
  
    return 0;  
}
```

- Conséquence : on ne peut pas faire référence au pointeur **this**

```
int Test::nbInstances() {  
    cout << this->x; x ←  
    return n;  
}
```

Puisque nbInstances peut être invoquée sans objet, le pointeur this n'est pas accessible

assert

- Macro **assert** : utilitaire facilitant le déboguage de programmes C++
 - Cette macro (exploitée par le précompilateur et le compilateur) permet d'interrompre l'exécution du programme si une condition est fausse
 - *Assert* signifie « valider l'hypothèse »
 - Lorsque le programmeur fait une hypothèse, il peut s'assurer qu'elle est toujours respectée via un `assert`

assert (suite)

- Exemple d'application

```
// Calcul de a % b. Attention: b ne doit pas être 0
int monModulo( int a, int b ) {
    return a - ( a / b ) * b ;
}
```

- Comment s'assurer que la fonction ne sera pas appelée avec 0 comme argument pour b
 - Vérifier et interrompre l'exécution

```
// Calcul de a % b. Attention: b ne doit pas être 0
int monModulo( int a, int b ) {
    if ( b == 0 )
        halt();    // Arrêter l'exécution

    return a - ( a / b ) * b ;
}
```

assert (suite)

- La stratégie du `halt()` est valide, mais elle ralentit l'exécution
 - La structure `if` est exécutée à chaque invocation de `monModulo`
 - Remplacer la structure par un **`assert`**

```
// Calcul de a % b. Attention: b ne doit pas être 0
int monModulo( int a, int b ) {
    assert( b != 0 ); // b ne doit pas être 0

    return a - ( a / b ) * b ;
}
```

- Si jamais `monModulo` est invoquée avec `b = 0`, alors un `halt()` sera automatiquement exécuté



assert (suite)

- Si **assert** vérifie aussi la condition et fait un `halt()`, quel est l'avantage ?
 - Tous les compilateurs C++ ont une option pour désactiver les `assert`
 - Ainsi, lorsque le programme aura été testé pour s'assurer qu'il n'invoque jamais `monModule` avec `b = 0`, on pourra désactiver les `assert` et le code exécutable ne contiendra plus ceux-ci
 - Si on utilise des structures `if` à la place des `assert`, le programmeur devra manuellement enlever toutes ces `if` (ou les commenter) afin de s'en débarrasser

assert (suite)

- Exemple de désactivation

- Utilisation de structures `if`

```
// Calcul de a % b. Attention: b ne doit pas être 0
int monModulo( int a, int b ) {
    // if ( b == 0 )
    //  halt();    // Arrêter l'exécution

    return a - ( a / b ) * b ;
}
```

- Utilisation de `assert`

- Désactiver l'option du compilateur puis recompiler (sans enlever les `assert` dans le code source)
 - La commande de précompilateur suivante désactive aussi les `assert` :

`#define NDEBUG`



Erreurs de programmation

- Invoquer une fonction membre non `const` sur un objet `const`
- Absence d'initialiseur de membre `const` dans un constructeur
- Utilisateur de l'opérateur point (.) sur une variable pointeur
 - L'utilisation de l'opérateur flèche (`->`) sur une variable d'instance statique ou sur un pointeur non initialisé à une instance dynamique
- Oublier de faire correspondre un `delete` à un `new`
- Référence au pointeur `this` dans une fonction membre statique
 - Ou déclarer `const` une fonction membre statique



Bonnes pratiques de programmation

- Déclarer `const` toutes les fonctions membres ne modifiant pas les attributs membres de l'instance
- Placer les relations d'amitié (`friend`) en premier dans la classe
 - Aucun identificateur d'accès n'est requis pour les prototypes `friend` car ils ne s'y appliquent pas
- Après avoir détruit une instance dynamique (avec `delete`), remettre le pointeur à 0

```
Date *p = new Date( 24, 2, 2009 );  
...  
delete p;  
p = 0;
```



Devoir #6

- Solutionnez le problème distribué par l'instructeur
 - Créer un nouveau projet console *Visual Studio C++*
 - Solutionner le problème tel que décrit, avec la spécification supplémentaire suivante
 - Le code source de votre projet doit être réparti dans trois fichiers : `CompteEpargne.h`, `CompteEpargne.cpp` et `Devoir_6.cpp`
 - L'instructeur vous fourni un programme principal (i.e. le fichier `Devoir_6.cpp`) qui vous permettra de tester votre classe **CompteEpargne**
 - N'oubliez pas les conventions d'écriture
 - Respectez l'échéance imposée par l'instructeur
 - Soumettez votre projet selon les indications de l'instructeur
 - **Attention** : respectez à la lettre les instructions de l'instructeur sur la façon de soumettre vos travaux, *sinon la note EC sera attribuée à ceux-ci*



Pour la semaine prochaine

- Vous devez relire le contenu de la présentation du chapitre 7
 - Il y aura un **quiz** sur ce contenu au prochain cours
 - À livres et ordinateurs fermés
 - Profitez-en pour réviser le contenu des chapitres précédents