Cours « IN202 »

Programmation Objet Orientée Objet
pour les Systèmes Embarqués

Ressources associées au Cours

- Wiki associé au cours
  - https://perso.ensta-paris.fr/~bmonsuez/Cours/doku.php?id=in202

- Mail de l'intervenant:
  - bruno.monsuez@ensta-paris.fr

Ce que ce Cours n’est pas !

- Un Cours présentant tout C++
  - Les éléments de syntaxe sont présentés.
  - Il est conseillé de se référer soit aux ouvrages conseillés, soit aux ressources en ligne référencée
- Un Cours de Programmation
- Un Cours d’Algorithmique

Les Objectifs de ce Cours

- Apprendre des Concepts de Programmation Embarqué et les apports de la programmation objet dans ce cadre
  - Phase 1: Notion d'Objets & Mise en œuvre en C++.
  - Phase 2: Programmation Embarquée en C++
  - Phase 3 : Mise en œuvre sur des cartes Arduino

Le Format du Cours

- Séance mélangeant Cours & Exercice sur Machine
- Evaluation par :
  - Un examen
  - Un projet sur environnement de simulation

Quelques notions

- « Système Embarqué » ou « Sytème Enfoui » ?
  - C’est la partie électronique qui commande un système
  - Pourquoi « Enfoui » ou « Embarqué » :
    - Pas systématiquement visible. Ex. ABS, Machine à Laver, Four, Système de Signalisation
    - Contribue à la fonctionnalité du système mais n’est pas l’essentiel du système.

Spécificités des Systèmes Embarqués

- Contraintes d'exploitation
  - Enveloppe Thermique, Consommation, Environnement électromagnétique…
- Interactions avec des capteurs/actionneurs
  - Dans ce cas, on parle de Système Cyberphysiques (ou Cyber Physical System)
- Souvent en interaction avec l'humain
  - On parle de Human Robot Interaction
- Parfois de pleine autorité ou autonome
  - Systèmes automatique ou systèmes autonomes

Composition des Systèmes Embarqués

- L’environnement matériel du Système
- La plateforme d’exécution des commandes
  - Plateforme matérielle électronique
  - Logiciel s’exécutant sur la plateforme matérielle électronique

Spécificités du logiciel embarqué

- Performances contraintes:
  mémoire (en Mo, voir Ko), puissance du processeur
- Temps-réel
  - Réponse à un évènement en un temps limité (moins de 200ms)
  - Capacité à traiter un certain nombre de données dans un temps limité (x images par secondes)
- Pas obligatoirement d'interfaces (écran/clavier/…)
  - Problème de mise au point
- Durée de fonctionnement/Mise-à-jour
- La qualité du code
  - Une erreur peut avoir des conséquences catastropiques

Programmez les Systèmes Embarqués

- Problème de ressources
  - Mémoire réduite
  - Performances réduites
- Résultat
  - Pas toujours d‘OS
  - Langage léger, efficace et compact
  - Pas toujours de suite complète de développement

Programmez les Systèmes Embarqués

- Etre capable de démontrer le bon comportement
  - Maitriser ce qui se passe
    - Au niveau fonctionnel
    - Au niveau temporel
    - Au niveau de la consommation des ressources

Le langage machine / L‘assembleur

Instructions propres au processeur

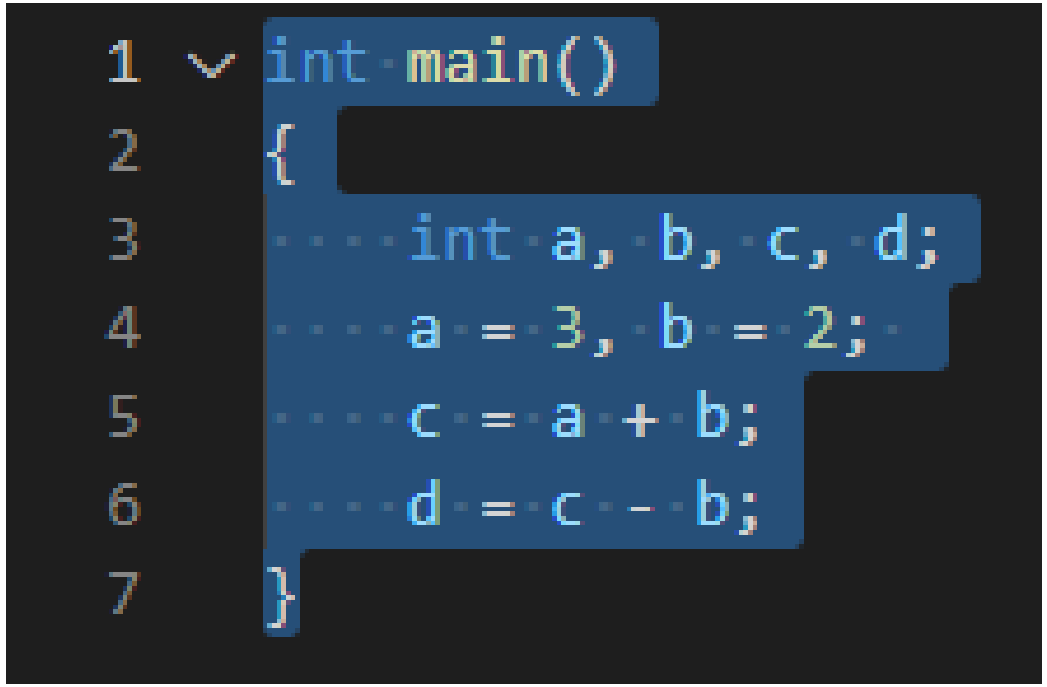
C = A + B; D = C – B;

Reg to Reg	Reg to Mem	Accumulator	Stack

load Reg1,A
load Reg2,B
add Reg3,Reg1,Reg2
store C,Reg3
load Reg1,C
load Reg2,B
sub Reg3,Reg1,Reg2
store D,Reg3
	load Reg1,A
	add Reg1,B
	store C,Reg1
	load Reg1,C
	sub Reg1,B
	store D,Reg1
		load A
		add B
		store C
		load C
		sub B
		store D
			push B
			push A
			add
			pop C
			push B
			push C
			sub
			pop D

Le langage C

- C est le langage historique de l'embarqué
  - Proche de l'assembleur
    - Traduction du code en assembleur presque immédiat
  - Portable
    - Compilateur C sur quasiment toutes les architectures



Avantage de C pour l‘embarqué

- **Le C est portable** : un même programme peut être compilé pour plusieurs microprocesseurs différents
- **Le C est plus haut niveau que l’assembleur** : les fonctions, tableaux et pointeurs sont des outils puissants absents de l’assembleur
- **Le C est bas niveau** tout de même : Le C offre un contrôle presque aussi fin des accès mémoire que l’assembleur.
- **Le C supporte l’intégration d’assembleur**

Limite du C pour l‘embarqué

- Faible productivité
  - Faible niveau d‘abstraction
- Sources de nombreuses erreurs
  - Pointeurs
  - Allocation mémoire
  - Transtypage
  - Pointeurs de fonctions…
- Faible réutilisabilité du code

Critique d‘un code C

```c
typedef struct _rational {
    int numerator;
    int denominator;
} rational

void add_rational(rational* result, rational* left, rational* right) {
    result->numerator = left->numerator * right->denominator
        + right->numerator + left->denominator;
    result->denominator =  left->denominator * right->denominator;
}

float rational_to_float(rational* value) {
    return (float)value->numerator / (float)value->denominator;
}

void main() {
    rational a = { 2, 3 }, b = { 4, 0 };
    rational* result;
    add_rational(result, &a, &b);
    printf(rational_to_float(&b));
}
```

Fonction devant être unique. Par de surchage

Pointeur pour passage par référence ! Possible de passer des structures non initialisées

Création d‘un nombre „rational“ inexistant (denominateur ne peut-être 0)

SegmentationFault !!!

Division By Zero !!!

Souhaits pour remplacer le C en embarqué

- Langage plus „sûr“
  - Fortement typé
  - Vérifiant des contraintes
  - Définissant des nouveaux types
  - Permettant de nommer les opérations plus naturellement
- Langage léger (en terme d’exécution)
  - Faible empreinte mémoire
  - Proche du hardware
  - Temps d’exécution prédictible

Plusieurs langages adaptés à l‘embarqué existent

- Ada => introduit la programmation modulaire
- C++ => apporte les objets à C
- Java => supporte les objets et la gestion automatique de la mémoire avec une machine virtuelle adaptée
- …

Une tendance de fonds : proposer le paradigme objet

TD

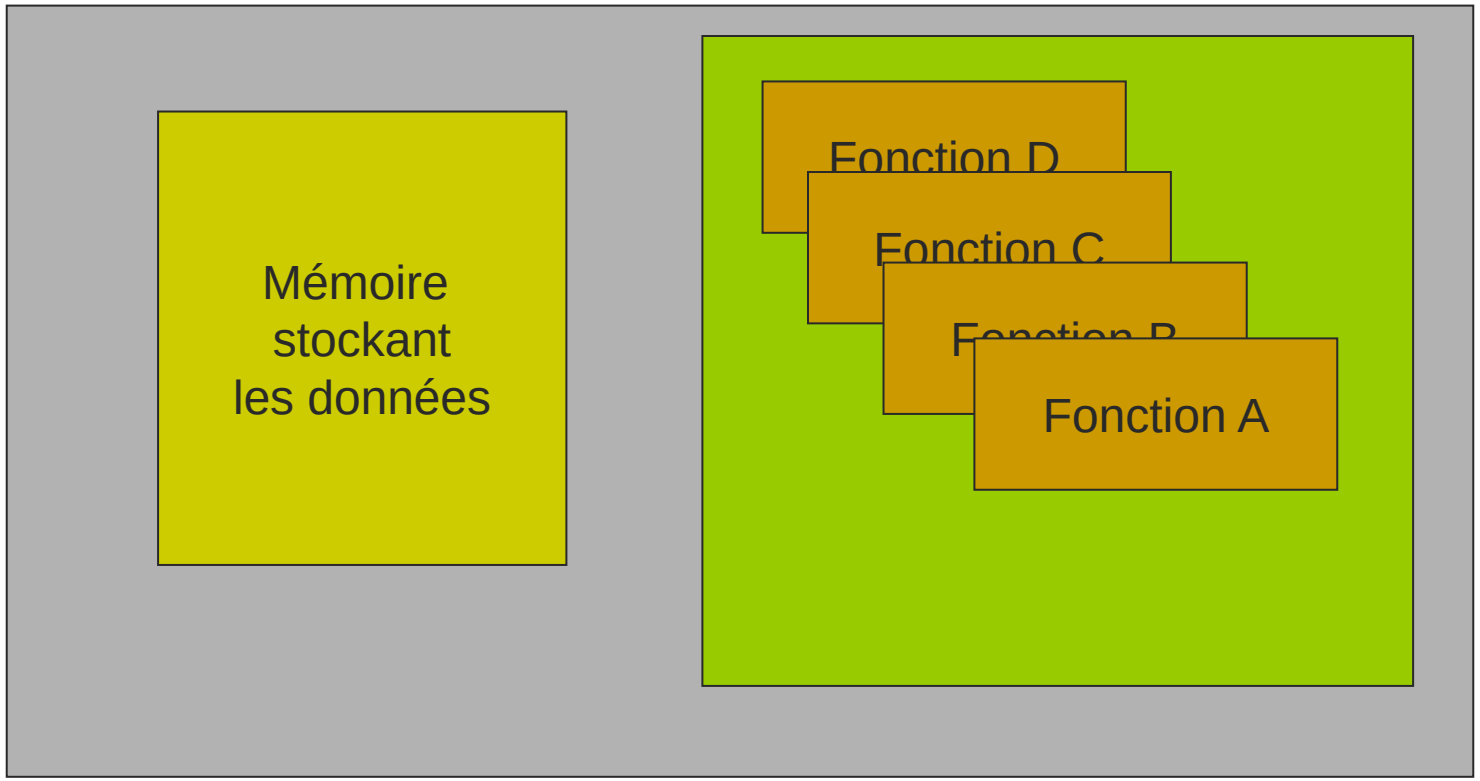
- Installation d‘un compilateur C++ + EDI
- Compilation du programme „rational“ en C++.

Pourquoi la notion d’objet

Introduction aux notions de classes et d’objets

D’où vient la notion d’objets ?

- Programme classique

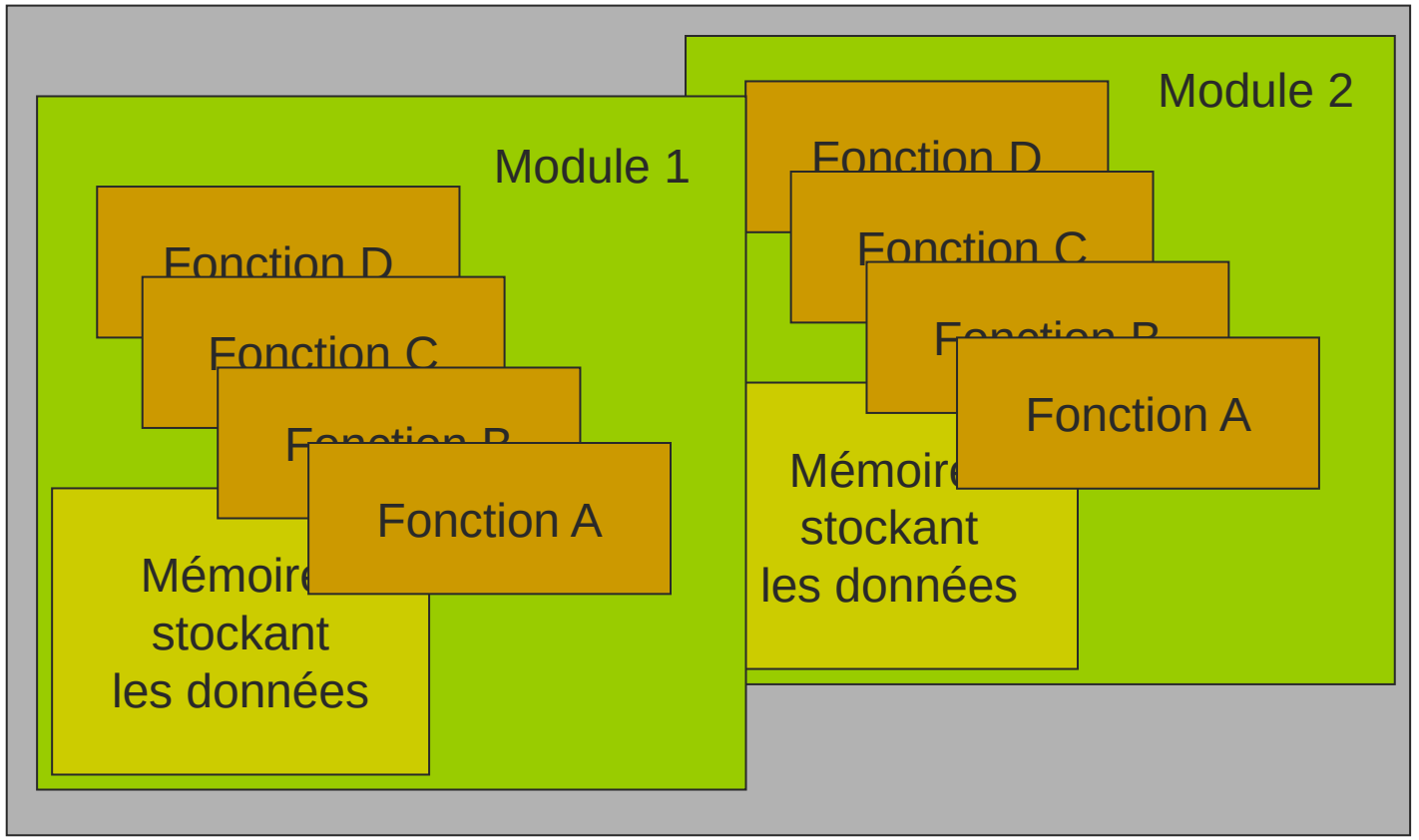


D’où vient la notion d’objets ?

- Défaut de la programmation structurée
  - Données séparées du « code »
  - Tout le monde peut accéder à toutes les données (sécurité, fiabilité, lisibilité)
  - Difficultés à définir des entités de « code » + « données » indépendantes et réutilisables
  - …
- Première solution
  - Modules :
    - Un module est un ensemble clos de fonctions et de variables.
    - Une interface défini les fonctions visibles à l’extérieur (Modula2, C/C++, Ada, …)

Pourquoi la notion d'objets ?

- Programmation modulaire



Pourquoi la notion d'objets ?

- Insuffisance des modules
  - Un programme a besoin de deux compteurs
    - Un compteur compte de 1 à 10
    - Un autre compteur compte de 1 à 30
  - Il faut écrire deux modules
    - Un pour le compteur comptant de 1 à 10
    - Un autre pour le compteur comptant de 1 à 30
  - Les modules
    - Ne permettent pas de définir un même code qui fonctionne pour des valeurs différentes

Qu’est ce qu’un objet ?

- Une classe est un « modèle » composé :
  - D’une description d’un ensemble de données
  - D’un ensemble de fonctions accédant aux données définies dans cette classe
- Un objet est une « instanciation » d’une classe
  - Chaque champ de données a une valeur particulière
  - Les actions pouvant être exécutées par l’objet sont définies par les fonctions de la classe

Un exemple de classe

```
struct MyCounter
{
        unsigned counter;
        unsigned max;

        unsigned getCounter() {
                return counter;
        }
        void increment() {
                counter ++;
                if(counter > max)
                                counter = 0;
        }
        void reset() {
                counter = 0;
        }
        void set(unsigned& value) {
                counter = (value <= max) ?
                                value : counter;
        void setMax(unsigned& value)
                 if(value >= max)
                                counter = 0:
                 max = value;
        }
};
```

Définition de deux champs

counter : valeur du compteur

max : valeur maximale du compteur

Définition de méthodes

reset() : remets le compteur à 0

set() et setMax() modifient les valeurs internes

increment() : incrémente le compteur

getCounter() : récupère la valeur du compteur

Un exemple d' utilisation des objets

```
void useObjectA() {

    MyCounter Counter1;
    MyCounter Counter2;

    Counter1.setMax(2);
    Counter2.setMax(4);

    Counter1.reset();
    Counter2.reset();

    for(unsigned i = 0; i <= 5; i++) {
      std::cout
                << "Valeur des compteurs ("
                << Counter1.counter
                << ", " << Counter2.counter
                << ")" << std::endl;
        Counter1.increment();
        Counter2.increment();
    }
}
```

- Instancie deux objets à partir de la classe MyCounter
- Le premier objet est un compteur entre 0 et 2, le deuxième objet un compteur en 0 et 4.
- Le code est commun au deux compteurs,
- Deux zones mémoires existent, l'une pour Counter1, l'autre pour Counter2.

TD : Implantation d'un Compteur

- Création d'un Projet Simple dans un environnement comme CodeBlocks ou MonoDevelop
- Créer un fichier .hpp contenant l'entête de l'objet compteur.
- Créer un ficher .cpp contenant le code de l'objet compteur.
- Tester le code l'objet.

Comment et où « instancier » un objet d’une classe

- Allouer de la mémoire pour stocker les données de l’instance
  - Sur le tas,
  - Sur la pile,
  - Dans la zone mémoire globale,

Créer un objet sur la pile (C++)

- Avantage
  - Allocation mémoire très rapide
  - Disparition automatique de l'objet en sortie de portée
- Inconvénient
  - Disparition automatique de l'objet en sortie de portée
- Remarque : c'est l'allocation effectuée dans l'exemple précédent

```
void myfunctionA() {
    MyCounter ACounter;
    std::cout <<
        ACounter.getCounter() << std::endl;
    std::cout <<
        ACounter.max << std::endl;
}
```

Créer un objet sur le tas

- Avantage
  - Plus souple que l'allocation sur la pile ou variable globale
  - Nombres infinis d'objets pouvant être créés.
- Inconvénient
  - Allocation mémoire lent
  - Accès plus lent
  - Problème de la destruction :
    - Soit de la responsabilité du programmeur (C++)
    - Soit le système analyse les objets qui ne sont plus référencés et les supprime (Java/C#)

```
void myfunctionB() {
  MyCounter* ACounter =
    new MyCounter();
  std::cout <<
   ACounter->getCounter() <<
              std::endl;
  std::cout << ACounter->max <<
              std::endl;
  …
  delete ACounter;
}
```

```
class Main {
 static void myfunctionB()
   {
      MyCounter ACounter  =
new MyCounter();
      ...
   }
}
```

TD : Création de l’objet compteur en C++

- Création d’un Projet Simple dans vote EDI
- Tester le code l’objet.

Distinction entre allocation et initialisation

- Deux phases durant la création d'un objet :
  - Allocation de la mémoire pour stocker les données associées à l'objet
  - Initialisation des valeurs par défaut des données définies par l'objet

- Comment initialiser les valeurs par défaut d'un objet :
  - L'opérateur « **new** », la déclaration comme variable globale ou locale
  - Une ou plusieurs méthodes appelées les constructeurs. *Ces méthodes ont le même nom que la classe en C++.*

- Le fonctionnement de la création d'un objet
  - Alloue la mémoire pour stocker l'objet
  - Procède à l'initialisation par défaut des données définies dans l'objet
    - Utilise le constructeur sélectionné
      Sinon utilise le constructeur par défaut s'il est défini,
      Sinon Initialise à 0 si l'allocation a lieu sur le tas
  - Procède enfin à l'appel du corps du constructeur pour initialiser l'objet.

Les constructeurs

- Les catégories de constructeur
  - *Les constructeurs explicites* : Le constructeur doit être appelé spécifiquement pour initialiser l'objet
  - *Les constructeurs implicites* : Le compilateur crée automatiquement un objet à partir d'un autre objet (conversion ou recopie) ou de rien du tout (constructeur par défaut)

- La syntaxe d'un constructeur
  - En C++

    MyClass(paramètres): data1(paramètres), data2(paramètres) …
    { corps du constructeur}

    data1 est un champ de MyClass de type dataType1 et on utilise le constructeur dataType1(paramètres) pour initialiser data1.

    explicit MyCounter(int valMax): counter(), max(valMax) {}
    ou
    explicit MyCounter(int valMax) { counter = 0; max = valMax }

Les constructeurs explicites

- Les constructeurs spécialisés

  - Prennent un certains nombres de paramètres
  - Soit complètent le constructeur par défaut, soit masquent le constructeur (si ce constructeur est non défini)
  - Ces constructeurs sont tous « explicites »

    MyClass(int aValue, const AnotherClass& anObject) {…}

Les constructeurs implicites

- Le constructeur par défaut
  - Ne prend pas de paramètres
    - En C++ si absent, un constructeur par défaut est automatiquement synthétisé.
    - En C++ l'objet est implicitement initialisé en utilisant ce constructeur.

      MyClass() {…}
- Les constructeurs de conversion
  - Ne prenne qu'un seul argument
  - Construit un objet automatiquement à partir de l'objet argument
  - Attention : mettre 'explicit' si le constructeur est un constructeur spécialisé

    MyClass(const AnotherClass& theSource) {…}
    MyClass(int aValue) {…}
- Les constructeurs de recopie
  - Ne prend comme argument une référence sur l'objet
  - Construit une copie de l'élément passé en référence

    MyClass(const MyClass& theSource) {…}

TD : Implantation de Constructeurs Spécialisés

La mort de l'objet

- La mort de l'objet se produit :
  - Quand le programmeur détruit l'objet (C++)
  - Quand l'on quitte la portée d'un objet défini sur la pile (C++)
  - Quand l'objet cesse d'être utilisé (C++ avec GC)

- La mort de l'objet intervient :
  - À un instant déterminé (**delete** ou sortie de la portée de définition)
  - À un instant indéterminé (quand on se rend compte que l'objet n'est plus utilisé)

Que se passe-t-il quand l'objet est détruit ?

- Avant de détruire un objet, certaines actions sont à réaliser
  - Libérer les ressources allouées
    - La mémoire, les fichiers ouverts, les ressources graphiques, etc
  - Informer éventuellement d'autres objets de sa disparition
    - Notification à un système de gestions d'objets
    - Mise à jour de compteurs d'instance d'une classe
    - …

Destructeur

- Une méthode spéciale est appelée avant la destruction de l'objet
  - En C++ (destructeur)
    - ~MyClass() {…}
    - Le destructeur est appelé au moment où l'objet est explicitement détruit.
    - La mémoire est immédiatement libérée après la destruction de l'objet.

Accéder aux champs et méthodes d’un objet

- La visibilité des méthodes et champs peut-être restreinte dans les objets :
  - Pour des raisons de sécurité
  - Pour des raisons de lisibilité
- Trois qualificateurs de visibilités :
  - Les méthodes et champs privées
  - Les méthodes et champs publics
  - Les méthodes et champs protégés
- Implantation
  - Définition de section débutant par **private: public:** ou **protected:**

Accès aux méthodes

- Les méthodes et champs privées :
  Accessibles uniquement à l'intérieur de la classe

- Les méthodes et champs publics :
  Accessibles de l'extérieur

- Les méthodes et champs protégés :
  Accessibles uniquement par les classes dérivant de la classe de base (voir prochain cours sur l'héritage)

TD : Sections publiques & Privées

Conclusions

Pourquoi un objet ?

- Créer une unité sémantique entre les données et le code
- Définir des stratégies d’allocation et de destruction d’un ensemble de composants.
- Rendre visible ou masque des méthodes et champs.