

LIFAPCD

CONCEPTION ET DEVELOPPEMENT D'APPLICATIONS

Université Claude Bernard Lyon 1
Licence Informatique 2^{ème} année
Nicolas Pronost

Programmation modulaire

Programmation modulaire

- Votre code est découpé en **modules**
 - Les classes appartenant au même module doivent être regroupées
 - Vue de l'extérieur : seules importent les fonctions publiques déclarées dans la déclaration, c.-à-d. la partie visible de l'iceberg
 - Règle d'intégrité : on ne touche pas directement aux données membres d'une classe, on passe par les fonctions membres
 - Permet l'abstraction de l'implémentation à l'utilisateur
 - Simplifie l'utilisation (manipulation par les fonctions prévues)
 - Facilite la recherche de bug (ex. points d'arrêt dans les fonctions)
 - Maintient la cohérence éventuelle entre les données membres
- Exemple : métaphore de la voiture
 - Pour conduire, un conducteur ne trifouille pas les câbles du moteur
 - Il utilise les « fonctions » `tournerVolant`, `changerVitesse`, etc.

Programmation modulaire

- Exemple : un module Voiture

Voiture.h

```
/* Directives pour éviter les inclusions multiples */
#ifndef VOITURE_H
#define VOITURE_H

/* Inclusions des entêtes utilisées */
#include "Moteur.h"
#include "Volant.h"
#include "Roue.h"

/* Définitions des classes (à usage externe au module) */
class Voiture {
    /* Les membres publiques d'abord, les privés ensuite */
public:
    Voiture();
    ~Voiture();
    void avancerVoiture(int acceleration);
    void tournerVolant(int degres);
    void changerVitesse(unsigned int nouvelle_vitesse);
    int getNiveauEssence() const;
    void fairePleinEssence();
};
```

Programmation modulaire

Voiture.h

```
private:
    /* Les données membres privées */
    Moteur leMoteur;
    Volant leVolant;
    Roue * lesRoues;

    /* Les fonctions membres privées */
    void trifouillerCable(int IDCable);

    //...

};

#endif /* Termine le #ifndef VOITURE_H */
```

Programmation modulaire

Voiture.cpp

```
/* Inclusion de l'entête du module */
#include "Voiture.h"

/* Inclusion d'entêtes standards */
/* Pour avoir sqrt(), cos(), sin(), ... */
#include <math.h>

/* Pour cin, cout, ... */
#include <iostream>

/* Définitions de constantes */
const float PI = 3.14159;
```

Programmation modulaire

Voiture.cpp

```
/* Définitions des fonctions membres publiques et privées */
Voiture::Voiture() {
    lesRoues = new Roue [4];
    //...
}

Voiture::~~Voiture() {
    delete [] lesRoues;
    //...
}

void Voiture::avancerVoiture(int acceleration) { ... }

//...

void Voiture::trifouillerCable(int IDCable) { ... }

//...
```

Avant de coder...

- Réfléchir aux modules (types de données qui vont être manipulés dans l'application) et aux liens entre les différents modules
 - ex: une voiture va contenir un moteur, un volant, ... et un moteur va contenir des pistons, etc.
- Réfléchir à l'organisation des classes en module
 - le plus souvent un module par classe
- Réfléchir aux fonctions **sans pour autant écrire le corps**
 - que vont elles faire?
 - que vont elles retourner?
 - quels paramètres vont elles prendre et en quel mode?
- Modéliser l'application sous forme **graphique** : nécessité d'avoir un modèle de **diagramme**

UML : Unified Modeling Languages

Comment s'y retrouver avec un programme comportant de nombreuses classes?

- **UML** est un formalisme permettant de définir et de visualiser un logiciel à l'aide de **diagrammes**
 - 13 types de diagramme existent
 - Combinés, les différents types de diagrammes UML offrent une vue complète des aspects statiques et dynamiques d'un logiciel
 - Nous ferons une utilisation très simplifiée d'UML avec 1 diagramme
 - Vous verrez plus d'UML en Génie Logiciel en Master 1
 - Remarque : UML n'est pas une **méthode de conception** (≠ AGILE etc.) mais un formalisme

Diagramme des classes

Comment s'y retrouver avec un programme comportant de nombreuses classes? → **Diagramme des classes**

- Chaque classe est décrite par une boîte contenant
 - La liste des données membres (nom, type et spécificateur d'accès)
 - La liste des fonctions membres avec
 - Spécificateur d'accès: privé (-) ou public (+) à la classe
 - Pour chaque fonction, la liste des paramètres (types et modes, optionnellement le nom) et le type de la valeur de retour
- Une boîte est en quelque sorte une entête indépendante du langage, une interface entre le code et son utilisation

Représentation des classes

NombreComplexe
+ Re, Im : réel
+ Fonction somme (IN NombreComplexe) → NombreComplexe + Fonction produit (IN NombreComplexe) → NombreComplexe + Procédure affiche () - Procédure reduire ()

Notations

Accès aux membres

+ : public → pour un usage externe à la classe

- : privé → pour un usage interne à la classe



Mode d'accès des paramètres

IN : donnée → paramètre en entrée (lecture)

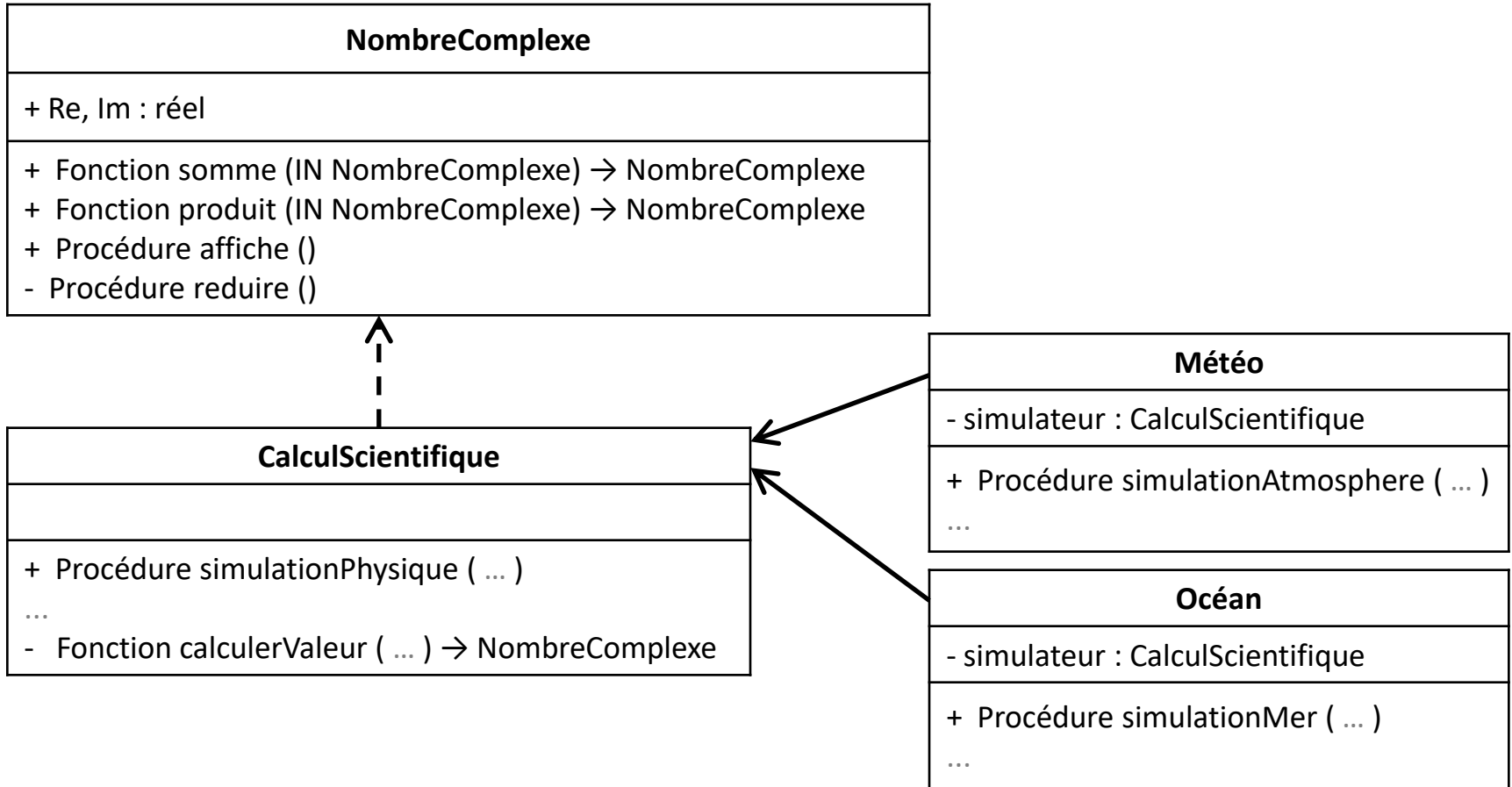
OUT : résultat → paramètre en sortie (écriture)

IN-OUT : donnée-résultat → paramètre utilisé en entrée/sortie (lecture/écriture)

Relations entre classes

- On utilise des flèches entre les boîtes pour indiquer le type de relation
- Il en existe de nombreuses en UML, vous en utiliserez dans cette UE que deux:
 -  La flèche pleine pour indiquer qu'une classe instancie un objet d'une autre classe
 - Ex: une donnée membre
 -  La flèche pointillée pour indiquer qu'une classe a besoin d'une autre classe (sans nécessiter d'en créer une instance)
 - Ex: un paramètre d'une fonction membre

Relations entre classes



Relations entre classes

- La classe B dépend de la classe A si B utilise au moins une fonction, une variable, ou un type déclaré dans A
- Les classes pointées par de nombreuses flèches constituent **le noyau du projet** → **à écrire (et tester) en premier**

Diagramme des classes

- Cas des pointeurs et références
 - Le diagramme des classes doit, autant que possible, être indépendant du langage (ex. éviter pointeur et référence spécifiques à C/C++)
 - Rôle des adresses et équivalent dans le diagramme
 - Passage de paramètre par adresse/référence à remplacer par le mode d'accès OUT ou IN-OUT dans le diagramme
 - Donnée membre représentant un lien vers un objet → indiquer explicitement le mot clé **lien**
 - Donnée membre utilisée ou destinée à être allouée en tableau (allocation dynamique) → indiquer explicitement le mot clé **tableau**
 - Un double pointeur (**) peut être un lien sur un tableau, un tableau de liens, un tableau 2D (matrice), un lien sur un lien → à préciser !

Diagramme des classes

- Cas des pointeurs et références
 - Exemple : classe en C++ et son équivalent dans le diagramme des classes

Code : Personne.h

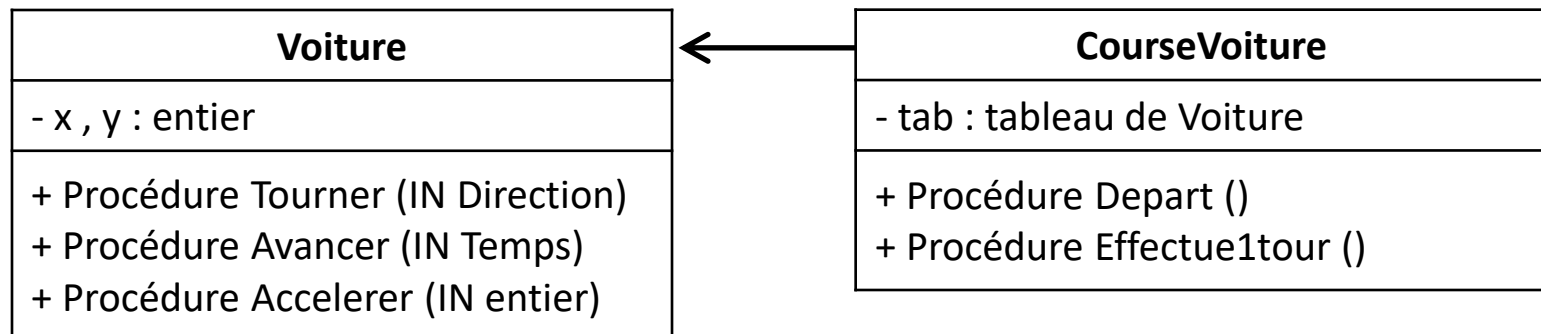
```
class Personne {  
    private:  
        string nom;  
        string prenom;  
        unsigned int age;  
        Personne * ptrPere;  
        Personne * ptrMere;  
        Personne ** tabPtrEnfants;  
    ...  
};
```

Equivalent diagramme

Personne
- nom, prenom : chaîne de caractères - age : entier naturel - ptrPere, ptrMere : lien sur Personne - tabPtrEnfants : tableau de liens sur Personne
...

Intégrité d'une classe

- Prendre la bonne habitude de « cacher » les données membres qui ne doivent pas être modifier directement
- On passe par l'intermédiaire de fonctions membres dédiées
 - *accesseurs* et *mutateurs* (get/set)
- Exemple : la classe CourseVoiture ne modifie pas directement les positions des voitures



Dépendances : cas de l'IHM

- Interface Homme Machine (IHM)
 - Menus déroulants, boîtes de dialogue, boutons, etc.
 - Souvent basée sur une librairie
- On doit pouvoir intervertir la librairie IHM avec des modifications mineures des modules non-IHM
 - Par exemple : passer d'un PC à un téléphone portable
- Conséquence
 - **Aucune classe** n'a besoin du module traitant l'IHM, c.-à-d. aucune flèche n'arrive sur ce module
 - Le module d'IHM peut avoir besoin de beaucoup de classes

Résumé : qu'est ce qu'on attend de vous ?

- Un cahier des charges initial
 - Fixant les fonctionnalités à atteindre
 - Ex: Je veux que mon application fasse ça, ça et ça
 - Découpant le projet en tâches
 - Il y aura T tâches
 - La tâche T1 sera considérée comme finie quand ceci marchera
 - La tâche T1 est effectué par l'étudiant E
 - La tâche T1 produira un livrable: ex. le module M
 - Agençant la réalisation des tâches dans le temps: diagramme de Gantt
- Programmation modulaire et diagramme de classes : à maintenir en continu pendant toute la durée du développement
- Quelques cycles de productions de l'application, avec à chaque fois
 - Définition et répartition des tâches et des tests à réaliser
 - Réalisation des tâches, tests, documentation (doxygen) et dépôt sur serveur (GitLab)

Règles de programmation

Règles de programmation

- Objectif : limiter le temps de mise au point et de débogage
- Quelques pistes
 - Fonction **assert**
 - Qualificatif **const**
 - Passage de paramètres
 - Création et destruction
 - Accesseurs (get) et mutateurs (set)

Question

Que se passe-t-il?

```
int div (int a, int b) {  
    return a/b;  
}  
  
int main() {  
    div(5,0);  
    return 0;  
}
```

```
int fact (int n) {  
    if (n==0) return 1;  
    else return n*fact (n-1);  
}  
  
int main() {  
    fact(-4);  
    return 0;  
}
```

Robustesse et verification: **assert**

```
#include <cassert>
int div (int a, int b) {
    assert(b!=0);
    return a/b;
}
```

```
#include <cassert>
int factorielle (int n) {
    assert(n>=0);
    //...
}
```

- Permet de vérifier une précondition de l'algorithme
 - Le programme **se termine** si la précondition n'est pas vérifiée
 - Assure une sortie « propre » en cas de problème
 - Court à écrire
- Message de sortie: **Assertion failed: b!=0, file main.cpp, line 3**
- Les assertions sont désactivées en mode release

A utiliser sans retenue

Passage de paramètres

- Le paramètre est une donnée (lecture seule)
 - Passage par valeur : `void affichage (Objet o)`
 - Recopie de l'objet sur la pile
 - => Peut être coûteux, à utiliser que pour les types primitifs
 - Passage par référence sur paramètre constant :
`void affichage (const Objet & o)`
 - Seule l'adresse mémoire est passée en paramètre
 - Interdiction de modifier l'objet référencé
 - => La solution idéale
- Le paramètre est une donnée-résultat
 - Passage par référence : `void effacer (Objet & o)`
 - => Pas d'autre choix

Fonction membre const

- Le mot clé **const** sert à indiquer qu'une fonction membre n'a pas le droit de modifier les données membres

```
class Position2D {  
    private:  
        float posX,posY;  
    public:  
        float distance () const {  
            return sqrt(posX*posX+posY*posY) ;  
        }  
};
```

- Ici la fonction distance est const. Elle n'a pas le droit de modifier les données posX et posY
- Une instruction telle que `posX = 2*posY;` ne compilerait pas
- Cela permet de s'assurer que les fonctions consultatives ne modifient jamais les données par erreur

Initialisation des données membres

- La construction d'un composé implique la construction de ses composants

```
class Roue { ... };  
class Cylindre { ... };  
class Batterie { ... };  
class Alternateur { ... };  
  
class Moteur {  
    Cylindre * tabCylindres [4];  
    Batterie bat;  
    Alternateur alt;  
    ...  
};  
  
class Voiture {  
    Moteur mot;  
    Roue * tabRoues;  
    int nbRoues;  
    ...  
};
```

```
Roue::Roue (...) { ... }  
Cylindre::Cylindre (...) { ... }  
Batterie::Batterie (...) { ... }  
Alternateur::Alternateur (...) { ... }  
  
Moteur::Moteur (...) : bat(...), alt(...) {  
    for (int iCyl=0; iCyl<4; iCyl++)  
        tabCylindres[iCyl] = new Cylindre(...);  
    ...  
}  
  
Voiture::Voiture (...) : mot(...) {  
    nbRoues = 4;  
    tabRoues = new Roue(...) [nbRoues];  
    ...  
}
```

- Bien sûr à organiser en plusieurs modules

Destruction des données membres

```
Voiture::~~Voiture () {  
    delete [] tabRoues;  
    ...  
}  
  
int main() {  
    Voiture v1;  
    Voiture * v2 = new Voiture(...);  
    ...  
    delete v2;  
    v2 = NULL;  
    ...  
    return 0;  
}
```

On doit appeler **le même nombre de new et de delete**.

99% de vos fuites mémoire seront des oublis de delete

- écrasement d'un pointeur avec un nouveau new ou une affectation
- oubli de libération en fin d'exécution du programme

- Ici le destructeur de v1 est automatiquement appelé en sortant du main
- Le destructeur de Voiture appelle automatiquement le destructeur de Moteur sur la donnée membre mot

Accesseur et mutateur

- A part dans *Voiture.cpp*, aucune fonction accède directement aux données de la classe *Voiture*
 - Conserve la cohérence qui peut exister entre plusieurs données (exemple simple : rayon et aire d'un cercle)
 - Permet de limiter les erreurs et de faciliter le débogage
 - Permet l'écriture de fonctions de haut niveau de manipulation de la classe
 - On peut changer l'implémentation sans changer l'interface... notion de boîte noire

Question : Accesseur et mutateur

- En imaginant que nbRoues est une donnée membre publique

```
int main() {  
    Voiture v;  
    v.nbRoues = 6;  
    /* nbRoue est modifiée directement sans changer le tableau */  
    v.affichePressionRoues();  
    return 0;  
}
```

Que va-t-il probablement se passer dans la
procédure **affichePressionRoues**?

Accesseur et mutateur

- Seul le module **voiture** sait ce qu'il faut faire si le nombre de roues change (ex. réallouer un tableau plus petit/grand, recopier des données, libérer l'ancien tableau, etc.)
 - On ne peut pas le court-circuiter
 - Nécessite des fonctions membres de manipulation des données membres
 - Les données membres sont donc privées et les fonctions de manipulation publiques
- Remarques
 - le plantage de l'exemple précédent peut intervenir longtemps après le changement de valeur de la donnée membre (ex. lors du premier accès à une adresse mémoire non allouée)
 - le compilateur ne fait **aucune vérification** des accès mémoire (nécessiterait l'exécution du programme...)
 - le développeur doit se discipliner

Accesseur et mutateur

- Les fonctions **set** sont appelées “mutateurs”

Voiture.cpp

```
void Voiture::setNbRoues (int nbr) {  
    /* Allocation d'un tableau à la nouvelle taille */  
    Roue * nvTab = new Roue [nbr];  
  
    /* Recopie des éléments dans le nouveau tableau */  
    memcpy(nvTab, tabRoues, sizeof(Roue) *nbRoues) ;  
  
    /* Mise à jour des données membres */  
    nbRoues = nbr;  
    delete [] tabRoues;  
    tabRoues = nvTab;  
}
```

main.cpp

```
int main() {  
    Voiture v;  
    v.setNbRoues(6);           /* met à jour tout l'objet v */  
    v.affichePressionRoues();  /* ne plante plus ! */  
    return 0;  
}
```

Question

- Que se passe-t-il?

```
int main() {  
    Voiture v;  
    cout << "pression roue numero 6 = ";  
    v.tabRoues[6].affichePression();  
    return 0;  
}
```

- En fonction de l'état de la mémoire
 - Le programme plante OU
 - Affichage 'étrange' (ex. caractère ASCII) OU
 - Affichage plausible (un entier) ce qui ne permet pas de détecter le bug immédiatement

Accesseur et mutateur

- Comme pour les mutateurs, seul le module **Voiture** sait ce qui est accessible ou non
 - Dans du code plus important, il est possible que ce bug passe inaperçu
- Les fonctions **get** sont appelées accesseurs
- Un **assert** peut aider à arrêter le programme lors d'un problème

```
int Voiture::getPressionRoue (int num_roue) const {  
    assert(num_roue >= 0 && num_roue < nbRoues);  
    return tabRoues[num_roue].getPression();  
}  
  
int main() {  
    Voiture v;  
    cout << "pression roue numero 6 = " << v.getPressionRoue(6);  
    /* Le programme s'arrête plus « proprement », avec un assert */  
    return 0;  
}
```

Accesseur et mutateur

- Remarquez que les accesseurs (get) sont toujours des fonctions membres **const**
 - leur but est de consulter l'état de l'objet uniquement
- Et les mutateurs (set) ne le sont jamais
 - leur but est de modifier l'état de l'objet
- Les accesseurs et mutateurs sont souvent imbriqués
 - lorsque les données membres sont des objets (type non primitif)
 - ex. Voiture::getPressionRoue appelle Roue::getPression
 - Roue::getPression va elle-même faire des vérifications (et/ou des calculs) et possiblement appeler un autre accesseur etc.

Accesseur et mutateur

- Attention, cela ne veut pas dire qu'il faut absolument mettre toutes les données membres privées
 - on veut souvent pouvoir manipuler directement une donnée

```
class Voiture {  
    public :  
        Moteur mot;  
  
        void setNbRoues (int nbr);  
        int getPressionRoue (int num_roue) const;  
  
    private :  
        Roue * tabRoues;  
        int nbRoues;  
};
```

```
#include "Voiture.h"  
  
int main() {  
    Voiture v;  
    v.mot.setPuissance(90);  
    cout << v.mot.getPuissance();  
    return 0;  
}
```

En vrac

- Indenter le code pour plus de lisibilité
- Ajouter des commentaires (préconditions, rôle d'une boucle etc.)
- Donner des noms explicites et au format homogène à vos variables, fonctions, classes, constantes etc.
- Créer des fonctions au lieu de copier/coller du code plusieurs fois
- Vérifier les désallocations de mémoire
- Gérer les erreurs
 - assertions (ou exceptions si vous connaissez)
 - codes d'erreur en retour de fonction
 - indicateurs passés en paramètre en mode donnée-résultat
- Optimiser votre code
 - itérations et récursions (surtout allocation/désallocation)
 - précalcul dans des tables et simplification des équations
 - utiliser les bons types