MOOC semaine 1

Classe, encapsulation et abstraction

en lien avec la programmation modulaire

Questions:

- En quoi une classe répond-elle aux objectifs de la programmation modulaire ?
- Y a-t-il encore un avenir pour struct ?

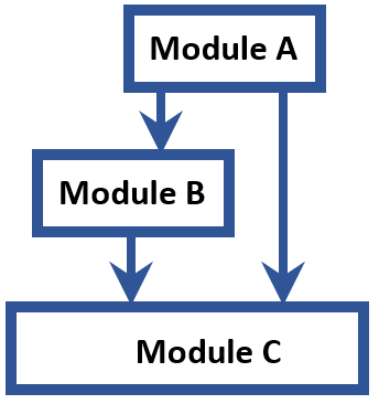
Plan:

- Interface et implementation à l'echelle d'une classe
- Portée et hiérarchie des espaces de noms
- Notion de type concret
- Le bon usage de struct comme type concret

Rappel (1)
Objectif de la programmation modulaire

Structurer un programme important

en entité fiables (***les modules***) …



… que l'on peut:

- mettre au point séparément
- intégrer pour constituer le programme
- éventuellement ré-utiliser pour d'autres programmes

[méthode de travail pour le développement de projet]

Rappel (2)
Mise en œuvre des grands principes

Abstraction: un module gère (en général) un type d'entité défini en termes de propriétés et d'actions dont il est responsable

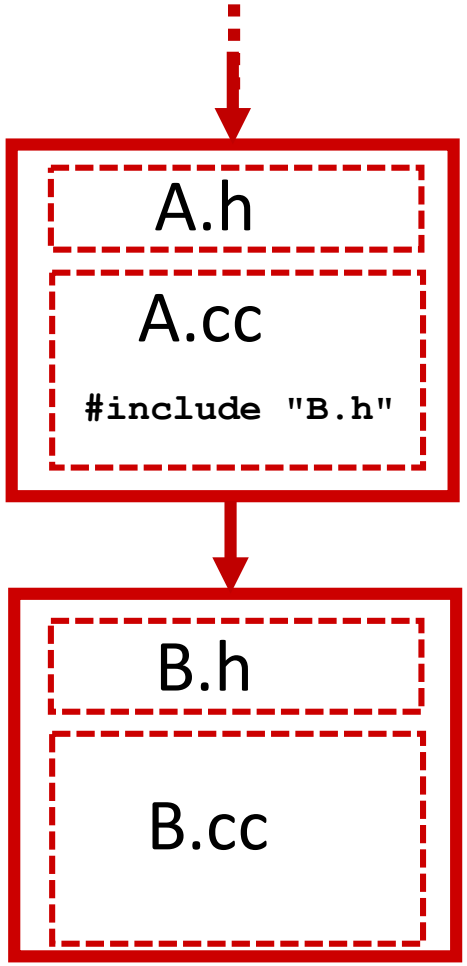
Encapsulation: le module se réserve le droit d'accéder *exclusivement* à certaines informations pour garantir son bon fonctionnement .

Séparation des fonctionalités/responsabilités: pour certaines tâche complexes, il est important d'identifier quel module en est responsable

Ré-utilisation: un module bien conçu peut être exploité dans une grande variété d'applications

Rappel (3)

un module = une interface + une implémentation



un **module** est composé de deux fichiers sources :

- Son fichier d' **interface** décrit son BUT ; il contient essentiellement les prototypes des fonctions *exportées* et documentées dans le fichier en-tête **(B.h)**.
- Son fichier d' **implémentation** est le code source définissant COMMENT les fonctions du module sont mises en œuvre (**B.cc**).

Seul le module contenant main() n'a pas d'interface car c'est le module de plus haut niveau qui est spécialisé pour une application donnée.

prog.cc

(unique) module de plus haut niveau contenant main()

```
#include <iostream>
#include "calcul.h"
using namespace std;

int main(void)
{
  int a(0), b(0);
  cin  >> a >> b ;
  cout << div(a,b) << end;
}
```

module **calcul** { **calcul.h**, **calcul.cc** }

calcul.h

```
int div(int num, int denom);
```

calcul.cc

```
#include "calcul.h"

int div(int num, int denom)
{
  if(denom != 0)
     return num/denom ;
  return 0;
}
```

Le concept de **classe** est l'offre du C++ pour répondre aux besoins exprimés par la programmation modulaire

Besoins: garanties de fiabilité, développement indépendant, ré-utilisation possible.

Abstraction: une **classe** gère un **type** d'entité défini en termes d'**attributs** et de **méthodes** dont elle est responsable

Encapsulation: **la classe** se réserve le droit d'accéder *exclusivement* à certaines éléments pour garantir son bon fonctionnement à l'aide des mots-clés :

- **public** *: accès possible en dehors de la classe*
- **private** *: accès restreint à la classe*

En quoi une **structure** / **struct** est-elle différente d'une **classe** / **class** ?

```
struct Date
{
    int day;
    int month;
    int year;
};

int main()
{
    Date birth;
    birth.day   = 1;
    birth.month = 3;
    birth.year   = 1998;
    …
```

Ces affectation sont autorisées car ***par défaut*** tous les champs d'une **struct** sont **public**

```
class Date
{
    int day;
    int month;
    int year;
};

int main()
{
    Date birth;
    birth.day   = 1;
    birth.month = 3;
    birth.year   = 1998;
    …
```

Ces affectation provoquent des erreurs de compilation car ***par défaut*** tous les champs d'une **class** sont **private**

Ne pas se laisser tenter de donner un accès **public** aux **attributs** d'une **class**

l'**Encapsulation** n'est pas possible avec une **structure** puisque tous ses champs peuvent être accédés et modifiés à partir du moment où on dispose du modèle de structure.

Le *DANGER* existe aussi pour une classe: le langage C++ laisse la liberté de mettre ce qu'on veut en accès **public**, aussi bien des **méthodes** (OK) que des **attributs** (très mauvaise pratique).

```
class Date
{
public:        ← BAD PRACTICE !
    int day;
    int month;
    int year;
};
```

Avec une classe, l'encapsulation n'est donc pas automatique.
La **responsabilité** de réaliser une **encapsulation** correcte revient à la *personne qui définit la classe*.

```
class Date
{
private:       ← GOOD START
    int day;
    int month;
    int year;
…
};
```

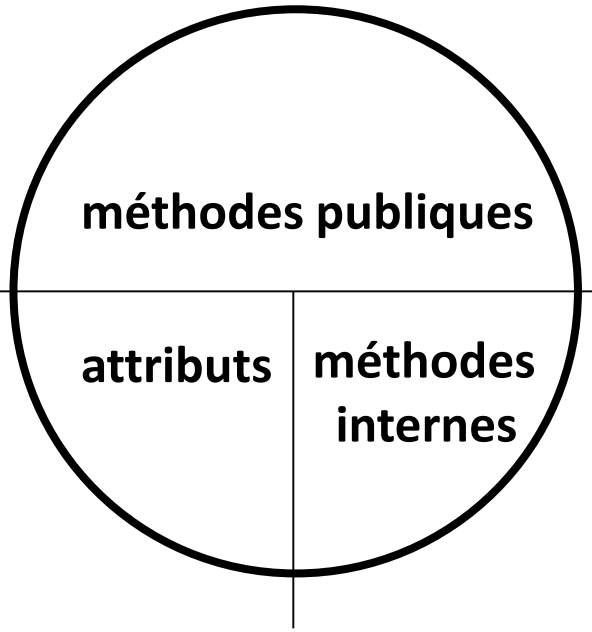
Mise en œuvre de **l'encapsulation** pour une **class**

Toutes les propriétés de la **class** représentées par des **attributs** doivent être en accès restreint = **private**.

Seul un ensemble de fonctions appelées **méthodes**, est utilisable à l'extérieur de la **class** grâce au mot-clé **public**.

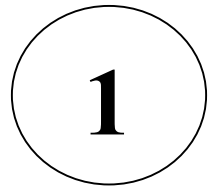
On appelle **interface** de la **class** l'ensemble des méthodes exposées avec **public**.

On appelle **implémentation** de la **class** l'ensemble des éléments **private**



Quel est le bon point de départ pour définir une classe ?

```
// définition de la classe    MOOC p10
class Rectangle {
public:
  // définition des méthodes
  double surface() const
    { return hauteur * largeur; }
  double getHauteur() const { return hauteur; }
  double getLargeur() const { return largeur; }
  void setHauteur(double h) { hauteur = h; }
  void setLargeur(double l) { largeur = l; }

private:
  // déclaration des attributs
  double hauteur;
  double largeur;
};
```



Montrer d'abord **l'interface** de la classe qui ne contient que des méthodes

Rassembler **l'implémentation** de la classe à la fin

En C++, il est autorisé de définir entièrement une méthode,
= écrire l'ensemble de ses instructions, dans
l'interface de la classe. Cependant -> cf slide suivant.

Pourquoi il faut ***externaliser*** la définition des méthodes

Les bonnes pratiques de la programmation modulaire cherchent *à minimiser les dépendances* entre les modules d'un programme.

2

MOOC p11

- Pour cela, **l'interface d'un module** doit être la plus légère possible
- Une interface légère est aussi plus lisible

Un module associé à une classe doit montrer **l'interface de la classe** dans **l'interface du module** (le fichier en-tête, par exemple rectangle.h) car elle contient les méthodes publiques.

Montrer les **prototypes** des méthodes publiques est nécessaire et suffisant pour les appeler dans un autre module

```
class Rectangle
{
public:
  // prototypes des méthodes
  double surface() const;

  // accesseurs
  double hauteur() const;
  double largeur() const;

  // manipulateurs
  void hauteur(double);
  void largeur(double);

private:
  // déclaration des attributs
  double hauteur_;
  double largeur_;
};
```

rectangle.h

Externalisation: la définition des méthodes avec l'opérateur de résolution de portée ::

Comme pour tout module, l'implémentation d'un module commence par inclure son fichier d'interface (fichier en-tête, .h)

La définition des méthodes étant en dehors du bloc de la **class** Rectangle, il faut préciser au compilateur que les méthodes appartiennent à cette classe.

L'opérateur de résolution de portée **::** est de priorité maximum

L'opérande gauche est un **espace de nom** = celui défini par la classe

Ici l'opérande droit est un nom de méthode de la classe

3

rectangle.cc

```
#include "rectangle.h"
double Rectangle::surface() const
{
   return hauteur_ * largeur_;
}

double Rectangle::hauteur() const
{
    return hauteur_;
}

... // idem pour largeur

void Rectangle::hauteur(double h)
{
    hauteur_ = h;
}

... // idem pour largeur
```

A propos des **espaces de nom** (**namespace**)

Chaque **class** définit un **espace de nom** autonome:

- À l'intérieur d'une **class** les **attributs** et les **méthodes** doivent avoir des noms distincts
- On peut utiliser les même noms d'attributs ou de méthodes dans différentes **class**

MOOC p11

```
class Rectangle
{
public:
  // prototypes des méthodes
  double surface() const;

  // accesseurs
  double hauteur() const;
  double largeur() const;

  // manipulateurs
  void hauteur(double);
  void largeur(double);

private:
  // déclaration des attributs
  double hauteur_;
  double largeur_;
};
```

Il existe deux autres espaces de noms importants en programmation modulaire :

- **L'espace de nom global**: une variable déclarée en dehors de toute fonction appartient à cet espace / tous les modules peuvent potentiellement y accéder.

INTERDIT pour le projet!!

- **Un espace de nom non-nommé** existe au niveau de **chaque module**. Une variable déclarée avec **static** en dehors de toute fonction est globalement visible au module mais pas en dehors. Idem pour une fonction.

Autorisé pour le projet

Comment déclarer/définir des variables/fonctions/etc dans **l'espace de noms non-nommé** ?

Syntaxe recommandée: déclaration/définition dans le bloc de **l'espace de noms non-nommé**

modul.h

```
// jamais dans l'interface !
```

modul.cc

```
#include "modul.h"
…
namespace {
    vector<int> tab;
    void f(int);
}
…
// usage interne seulement
   f(tab[0]);
…

namespace { // définition des fonctions internes
    void f(int n){
        …
    }
}
```

Alternative: Déclaration/définition avec **static** en dehors de tout bloc

modul.h

```
// jamais dans l'interface !
```

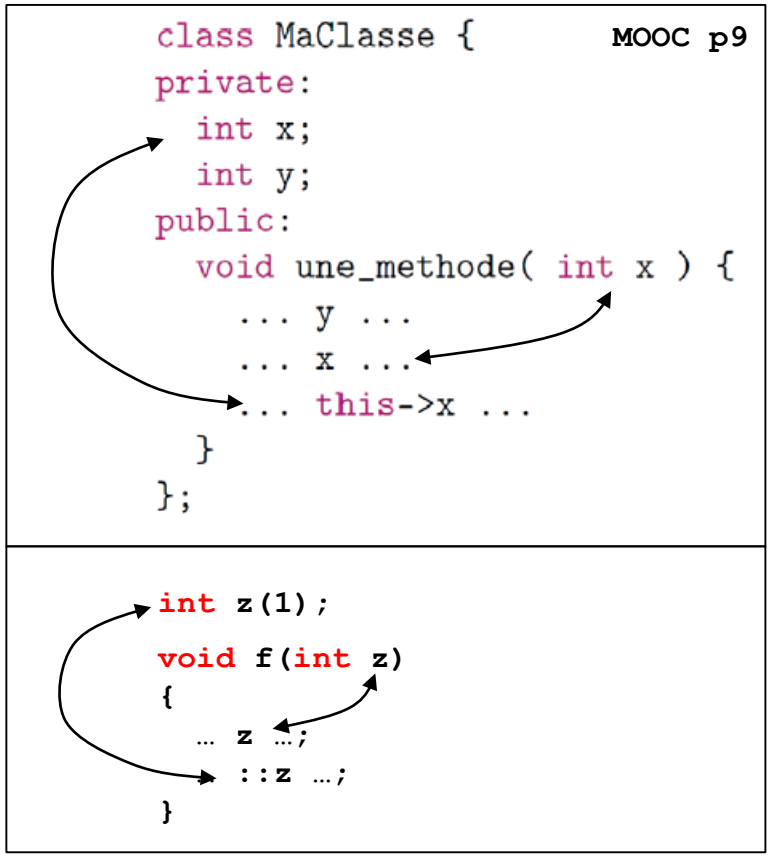
modul.cc

```
#include "modul.h"
…
static vector<int> tab;
static void f(int);

…
// usage interne seulement
   f(tab[0]);
…


static void f(int n){
   …
}
```

Rappel sur le masquage / le pointeur **this**

Une variable déclarée localement ***masque*** une variable de même nom déclarée à un niveau plus global.

S'il s'agit d'un paramètre de méthode qui masque un attribut, on peut utiliser le pointeur **this** qui mémorise l'adresse de l'instance sur laquelle est appelée la méthode.



Si une variable locale z masque une variable z de l'espace de nom global, l'écriture ::z désigne explicitement la variable de l'espace de nom global.

Il n'existe aucun moyen pour contourner le masquage d'une variable de l'espace de nom non-nommé. Il faut éviter d'utiliser ce nom localement.

Notion de type **concret**

Un type **concret** est un type dont le **compilateur** dispose de la description détaillée, ce qui lui permet de **calculer l'espace mémoire** nécessaire pour une variable de ce type.

Un type **concret** permet de **déclarer** une variable de ce type

Exemple1 : les types de base
int, double, bool, char

Exemple2 : les types définis avec un **modèle de structure** complet

Si le compilateur ne connait pas les détails du modèle de structure il produit une ***erreur***, comme ici pour **d1**
Aucun problème pour **d2** car le modèle a été fourni entre temps

```
struct Date ; //prédéclaration

Date d1={1,1,1000};

struct Date
{
    int day;
    int month;
    int year;
};

Date d2={2,2,2000};
```

La définition d'une classe définit un type concret car le compilateur connait les types de l'ensemble des attributs.

Le bon usage de **struct** comme type **concret**

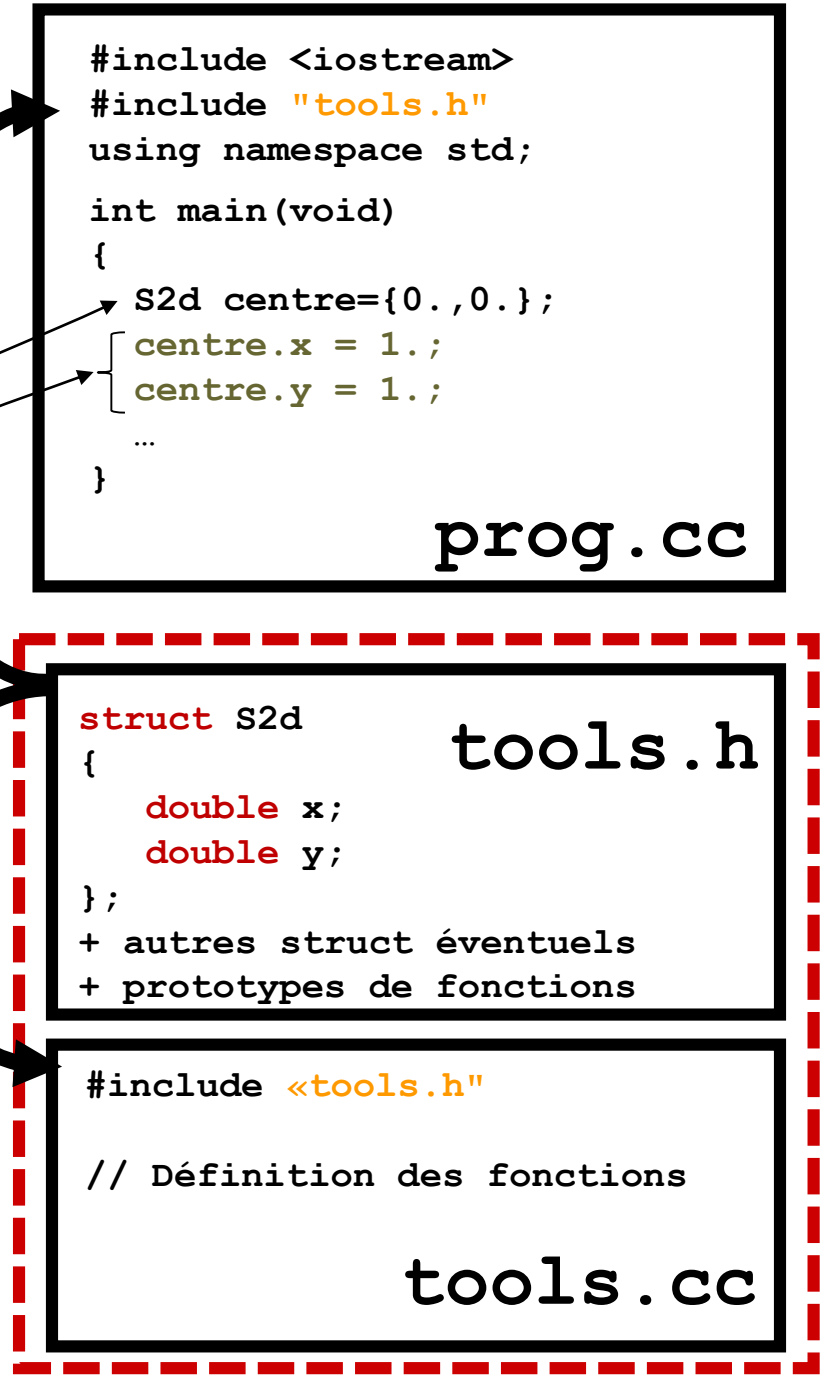
Type concret <=> le modèle de **struct** est exporté dans l'interface

Il est pertinent d'exploiter **struct** pour des types concrets de **bas-niveau** pour lesquels les risques d'utilisations incorrectes sont faibles et l'interface est stable.

Supposons qu'on veuille définir un module générique de bas-niveau avec des types concrets permettant de travailler avec des points, vecteurs, cercle, etc..

Dans cet exemple ce module s'appelle **tools** ; on y définit un type **S2d** qui permet de représenter aussi bien un point qu'un vecteur dans le plan 2D.

Grâce au type concret, on peut déclarer des variables de ce type et accéder/modifier librement les champs.



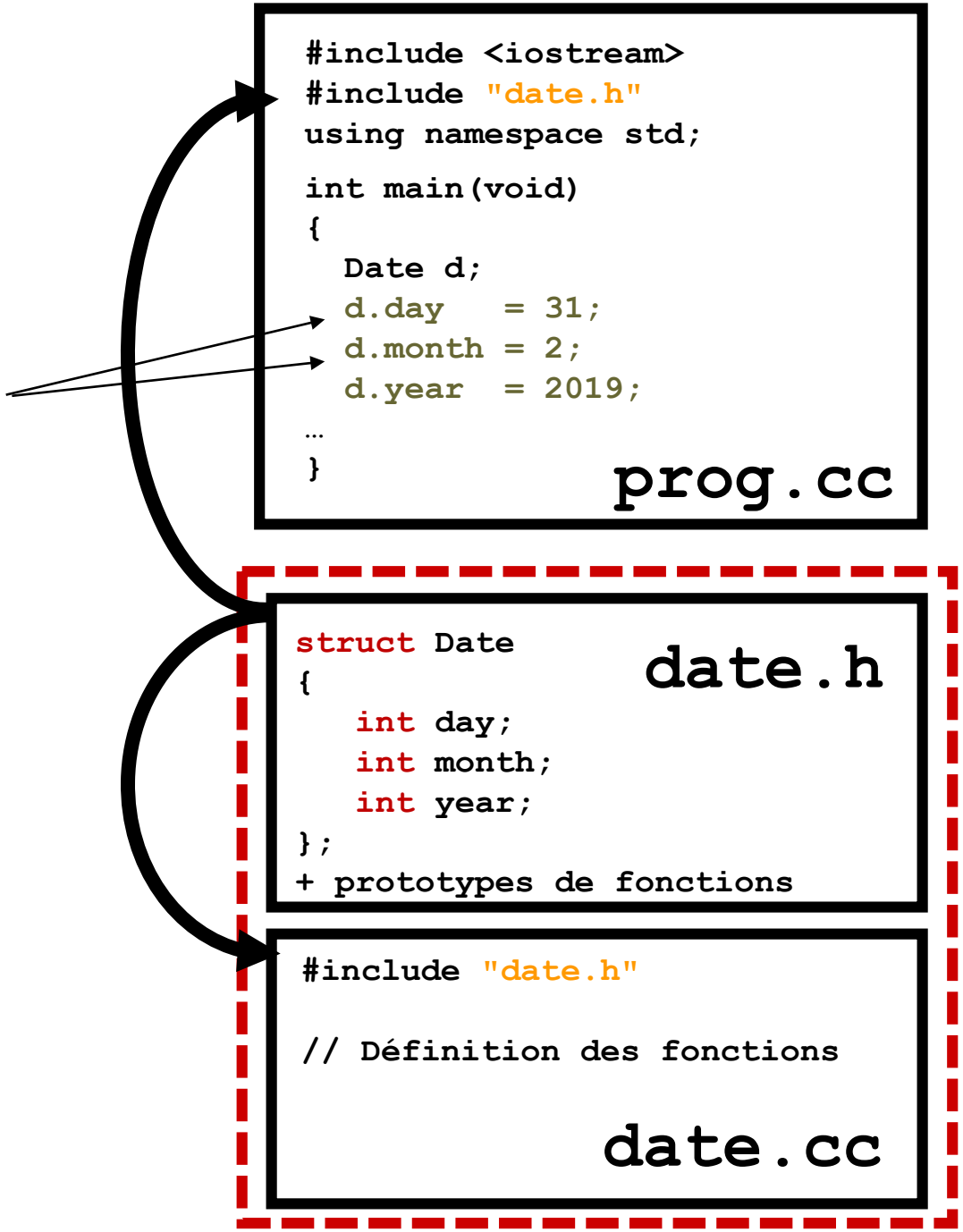
Usage à éviter: lorsqu'un contrôle fin est requis pour garantir la *validité* du type **concret**

Le module prog.cc peut affecter des valeurs quelconques à une variable de type **Date**.

Cela veut dire que le module **date** perd le contrôle sur la validité des instances de type **Date**.

Dans ce cas, c'est un mauvais choix d'utiliser une structure et de rendre le modèle de la structure visible dans l'interface du module pour le type **Date**.

Une classe est préférable.



SpeakUp Room 93621

Voici une mise en œuvre d'une classe **Datum** pour gérer une date de calendrier.

Question: est-elle conforme *aux buts* du principe **d'encapsulation** ?

A: Oui

B: Non

datum.h

```
class Datum{
public:
    void getDatum(int& d, int& m, int& y) const;
    void setDatum(int  d, int  m, int  y);
    void printDatum() const;
private:
     int day;
     int month;
     int year;
};
```

datum.cc

```
#include <iostream>
#include "datum.h"
using namespace std;

void Datum::getDatum(int& d, int& m, int& y) const{
        d = day;  m = month;  y = year;
}

void Datum::setDatum(int  d, int  m, int  y){
        day = d;  month = m;  year = y;
}
void Datum::printDatum() const{
        cout << "Nous sommes le " << day
             << " du mois "       << month
             << " de l'année "    << year << endl;
}
```

Résumé

- **Les Principes de la programmation modulaire sont efficacement mis en oeuvre en C++ avec le concept de class.**
- **Une class est constituée d'une interface (la partie public) et d'une implémentation (la partie private).**
- **Il ne faut pas rendre les attributs public**
- **une bonne correspondence entre *module* et class implique *d'externaliser* la définition des méthodes dans l'implémentation du module qui lui est associé.**
- **L'espace de noms *non-nommé* est pertinent pour déclarer un petit nombre de variables globale à un module avec static**
- **un type *concret* défini avec struct doit être réservé pour des entités de bas-niveau avec faible risque d'erreur.**