

MOOC semaine 1

Classe, encapsulation et abstraction *en lien avec la programmation modulaire*

Questions:

- comment **class** répond aux objectifs de la programmation modulaire ?
- Y a-t-il encore un avenir pour **struct** ?

Plan:

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

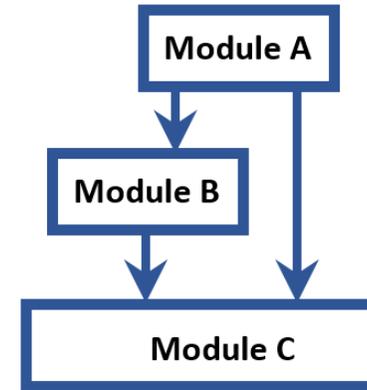
Rappel (1)

Objectif de la programmation modulaire

Structurer un programme important
en entité fiables (*les modules*) ...

... que l'on peut:

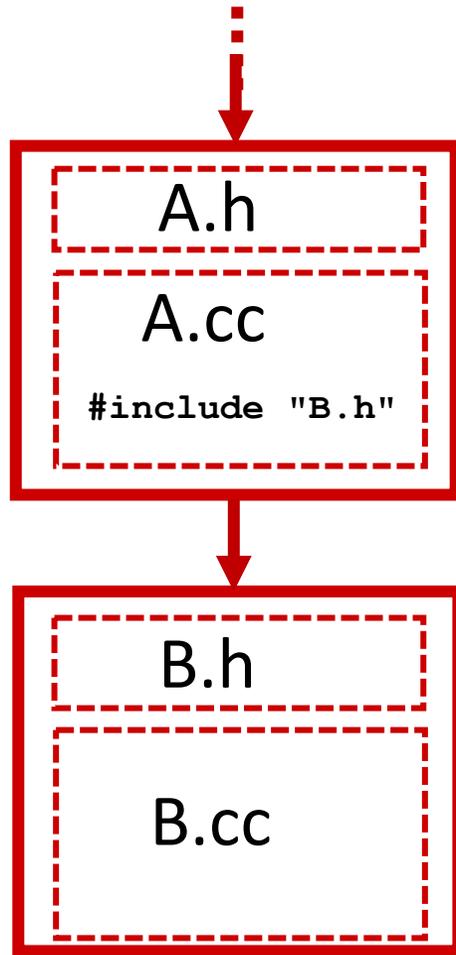
- mettre au point séparément
- intégrer pour constituer le programme
- éventuellement ré-utiliser pour d'autres programmes



[section 3 de méthode de travail pour le développement de projet]

Rappel (2)

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

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

```
#include "calcul.h"

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

Ce code produit un exécutable :

```
g++ -std=c++11 calcul.cc -c
g++ -std=c++11 prog.cc -c
g++ prog.o calcul.o -o prog
```

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

Terminologie: un synonyme de **méthode** est **fonction membre** d'une classe,
par opposition à une **fonction libre** (*free function*) indépendante d'une classe

Encapsulation: la **classe** se réserve le droit d'accéder *exclusivement* à certains 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

Quizz1: différence entre **structure** / **struct** et **classe** / **class**

code1

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

int main()
{
    Date birth;
    birth.day    = 30;
    birth.month  = 2;
    birth.year   = 2025;
    ...
}
```

code2

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

int main()
{
    Date birth;
    birth.day    = 1;
    birth.month  = 3;
    birth.year   = 2025;
    ...
}
```

- A: **code1** et **code2** compilent sans problème
- B: seul **code1** compile sans problème
- C: seul **code2** compile sans problème
- D: aucun des deux ne compile correctement

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).

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

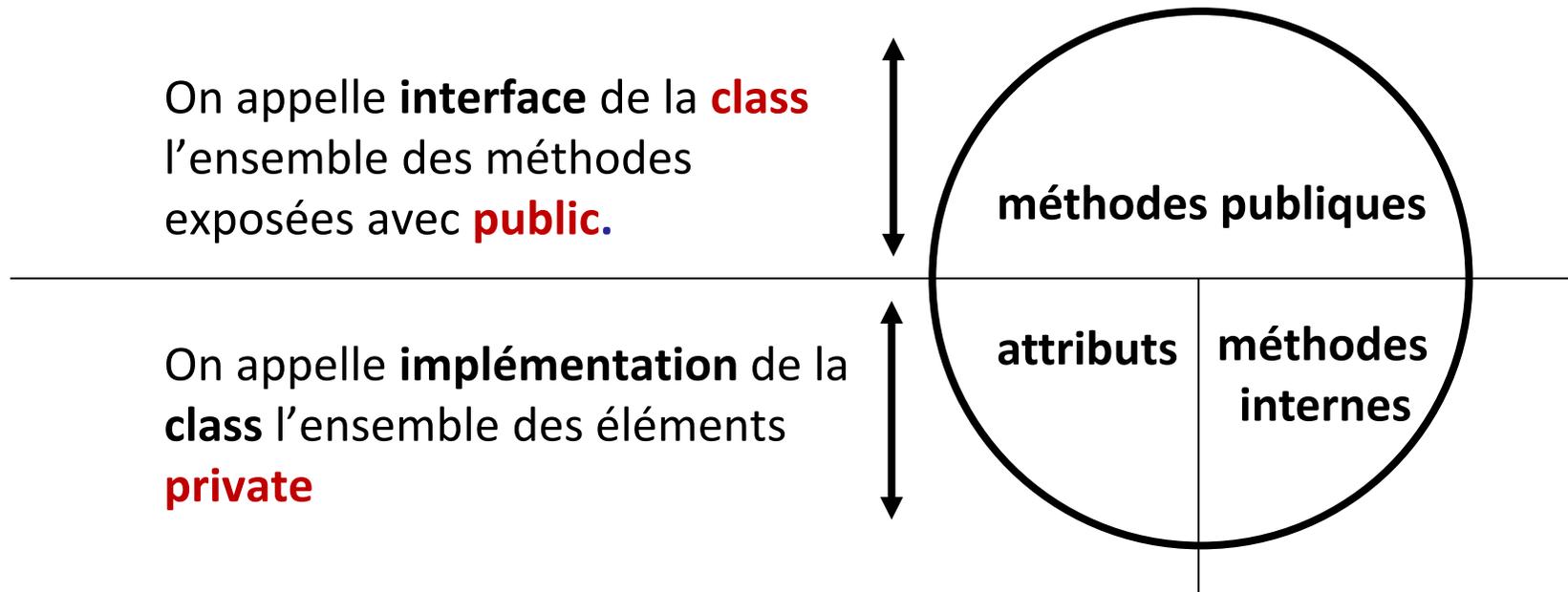
```
class Date
{
private: ← GOOD START
    int day;
    int month;
    int year;
};
```

```
...
};
```

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



Comment définir une classe d'une manière lisible ?

```
// 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;
};
```

1

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, c'est à dire écrire l'ensemble de ses instructions, dans **l'interface de la classe**. Cependant c'est peu lisible, donc à éviter (on *externalise* la définition des méthodes comme présenté plus loin)

Une précision importante sur le concept de **Portée de classe**

La définition d'une **class** est faite à l'intérieur d'un bloc qui constitue la **portée de la classe**.
Tous les éléments déclarés à l'intérieur de la portée de la classe sont connus dans toute la classe **dès le début du bloc** de la classe.

```
// définition de la classe
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;
};
```

MOOC p10

Cela explique que les **attributs** peuvent être utilisés par les **méthodes** même si les attributs sont déclarés après les méthodes.

Quizz2: différence entre la portée «classique» des blocs { } et la portée définissant une **class**

code1	{	<pre>int main() { int a(99); double x(a); double y(b); int b(33); ... }</pre>	}	code2
		<pre>int main() { int a(99); double x(a); int b(33); double y(b); ... }</pre>		

- A:** **code1** et **code2** compilent sans problème
- B:** seul **code1** compile sans problème
- C:** seul **code2** compile sans problème
- D:** aucun des deux ne compile correctement

Pourquoi il faut *externaliser* la définition des méthodes

2

MOOC p11

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

- 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: définition des méthodes en indiquant qu'elles appartiennent à *l'espace de noms* de la classe grâce à l'opérateur de résolution de portée ::

3

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 **l'espace de nom** défini par le type créé par la classe => **Rectangle**

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

rectangle.cc

```
#include "rectangle.h"

double Rectangle::surface() const
{
    return hauteur_ * largeur_;
}

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

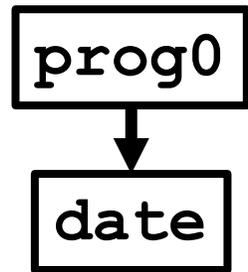
... // idem pour largeur

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

... // idem pour largeur
```

Quizz3: ce programme modulaire

architecture
logicielle



```
prog0.cc
#include <iostream>
#include "date.h"

using namespace std;

int main()
{
    Date d_ref = reset();

    print(d_ref);

    return 0;
}
```

```
date.h
struct Date
{
    int day;
    int month;
    int year;
};

void print(Date d);

// return the date of the first day
// of the Gregorian calendar: 25.02.1582
Date reset();
```

```
date.cc
#include <iostream>
#include "date.h"

using namespace std;

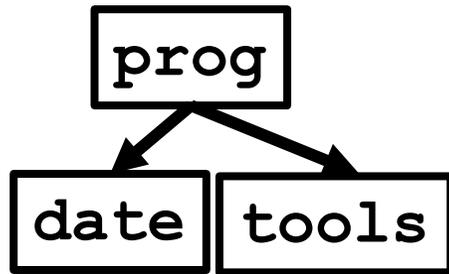
void print(Date d)
{
    cout << "Nous sommes le " << d.day
         << " du mois " << d.month
         << " de l'année " << d.year << endl;
}

Date reset()
{
    return {25, 2, 1582};
}
```

- A: ... produit un exécutable
- B: ... compile séparément
mais il y a un problème à l'édition de liens
- C: ... produit une erreur de compilation pour prog.cc
- D: ... produit une erreur de compilation pour date.cc

Quizz4: ce programme modulaire

architecture
logicielle



```
prog.cc
#include <iostream>
#include "tools.h"
#include "date.h"

using namespace std;

int main()
{
    Date d_ref ;

    print(d_ref);

    return 0;
}
```

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

void print(Date d);

// return the date of the first day
// of the Gregorian calendar: 25.02.1582
Date reset();
```

*même module
que pour Quizz3*

```
#include <iostream>
#include "date.h"

using namespace std;

void print(Date d)
{
    cout << "Nous sommes le " << d.day
          << " du mois " << d.month
          << " de l'année " << d.year << endl;
}

Date reset()
{
    return {25, 2, 1582};
}
```

```
struct S2d
{
    double x;
    double y;
};

void print(S2d s);
```

tools.h

```
#include <iostream>
#include "tools.h"

using namespace std;

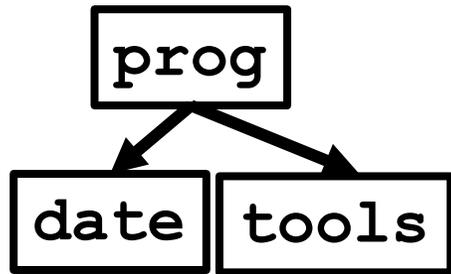
void print(S2d s)
{
    cout << "x = " << s.x << " y = "
          << s.y << endl;
}
```

tools.cc

- A: ... produit un exécutable
- B: ... compile séparément
mais il y a un problème à l'édition de liens
- C: ... produit une erreur de compilation pour prog.cc
- D: ... produit une erreur de compilation pour tools.cc

Quizz5: ce programme modulaire

architecture
logicielle



prog.cc

```
#include <iostream>
#include "tools.h"
#include "date.h"

using namespace std;

int main()
{
    Date d_ref ;

    print(d_ref);

    return 0;
}
```

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

void print(Date d);

// return the date of the first day
// of the Gregorian calendar: 25.02.1582
Date reset();
```

*même module
que pour Quizz3*

```
#include <iostream>
#include "date.h"

using namespace std;

void print(Date d)
{
    cout << "Nous sommes le " << d.day
          << " du mois " << d.month
          << " de l'année " << d.year << endl;
}

Date reset()
{
    return {25, 2, 1582};
}
```

```
struct S2d
{
    double x;
    double y;
};

void print(S2d s);

// return a struct
// filled with zeros
S2d reset();
```

tools.h

```
#include <iostream>
#include "tools.h"

using namespace std;

void print(S2d s)
{
    cout << "x = " << s.x << " y = "
          << s.y << endl;
}

S2d reset()
{
    return {0,0};
}
```

tools.cc

- A: ... produit un exécutable
- B: ... compile séparément
mais il y a un problème à l'édition de liens
- C: ... produit une erreur de compilation pour prog.cc
- D: ... produit une erreur de compilation pour tools.cc

Programmation modulaire et espace de noms: **namespace**

Observation : l'intégration des nombreux modules objets dans l'exécutable par l'éditeur de liens peut faire apparaître des **collisions de noms** (de fonctions) alors que l'étape de compilation n'en a pas détectées.

Diagnostic: duplication du même prototype de fonctions dans plusieurs modules. les modules compilent séparément tant qu'il n'y a pas de duplication à l'échelle du module individuel.

Remède: introduire un espace de noms distinct avec le mot clef **namespace** pour chaque module qui exporte des fonctions libres (*free functions*) c'est à dire liées à aucune classe.

syntaxe .h :

```
namespace exemple
{
    struct Exemple
    {
        // ici des champs de struct
    };

    // prototype exporté

    void print(Exemple e);

    // return a struct Exemple
    Exemple reset();
}
```

syntaxe .cc :

```
#include <iostream>
#include "exemple.h"

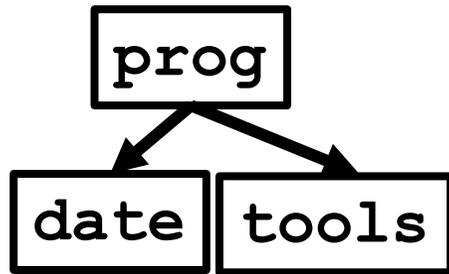
using namespace std;

namespace exemple
{
    void print(Exemple e)
    {
        // affichage des champs de cette struct ;
    }

    Exemple reset()
    {
        // return {avec valeur désirée des champs};
    }
}
```

Quizz6: ce programme modulaire

architecture
logicielle



```
prog1.cc
#include <iostream>
#include "tools1.h"
#include "date.h"

using namespace std;

int main()
{
    Date d_ref ;
    print(d_ref);
    return 0;
}
```

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

void print(Date d);

// return the date of the first day
// of the Gregorian calendar: 25.02.1582
Date reset();
```

*même module
que pour Quizz3*

```
#include <iostream>
#include "date.h"

using namespace std;

void print(Date d)
{
    cout << "Nous sommes le " << d.day
    << " du mois " << d.month
    << " de l'année " << d.year << endl;
}

Date reset()
{
    return {25, 2, 1582};
}
```

```
namespace tools
{
    struct S2d
    {
        double x;
        double y;
    };

    void print(S2d s);

    // return a struct
    // filled with zeros
    S2d reset();
}
```

tools1.h

```
#include <iostream>
#include "tools1.h"

using namespace std;

void tools::print(tools::S2d s)
{
    cout << "x = " << s.x << " y = "
    << s.y << endl;
}

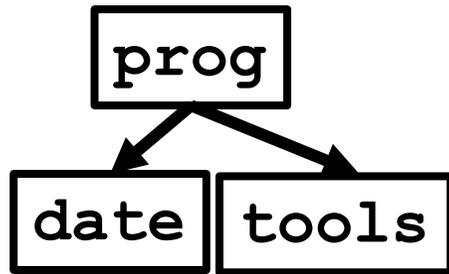
tools::S2d tools::reset()
{
    return {0,0};
}
```

tools1.cc

- A: ... produit un exécutable
- B: ... compile séparément
mais il y a un problème à l'édition de liens
- C: ... produit une erreur de compilation pour prog.cc
- D: ... produit un erreur de compilation pour tools.cc

Quizz7: ce programme modulaire

architecture
logicielle



```
prog1.cc
#include <iostream>
#include "tools1.h"
#include "date.h"

using namespace std;

int main()
{
    Date d_ref ;
    print(d_ref);
    return 0;
}
```

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

void print(Date d);

// return the date of the first day
// of the Gregorian calendar: 25.02.1582
Date reset();
```

*même module
que pour Quizz3*

```
#include <iostream>
#include "date.h"

using namespace std;

void print(Date d)
{
    cout << "Nous sommes le " << d.day
          << " du mois " << d.month
          << " de l'année " << d.year << endl;
}

Date reset()
{
    return {25, 2, 1582};
}
```

```
namespace tools
{
    struct S2d
    {
        double x;
        double y;
    };

    void print(S2d s);

    // return a struct
    // filled with zeros
    S2d reset();
}
```

tools1.h

```
#include <iostream>
#include "tools1.h"

using namespace std;

namespace tools
{
    void print(S2d s)
    {
        cout << "x = " << s.x << " y = "
              << s.y << endl;
    }

    S2d reset()
    {
        return {0,0};
    }
}
```

tools1.cc

- A: ... produit un exécutable
- B: ... compile séparément
mais il y a un problème à l'édition de liens
- C: ... produit une erreur de compilation pour prog.cc
- D: ... produit une erreur de compilation pour tools.cc

Le point sur les **espaces de nom (namespace)**

1) Par construction 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**

```
class Rectangle      MOOC p11
{
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_;
};
```

2) Pour éviter des collisions de noms provenant d'autres modules dans un programme, chaque **module peut** définir un **espace de nom nommé** avec **namespace** (ex: quizz avec tools1)

*Autorisé
pour le projet*

3) De plus, chaque **module peut** rendre confidentielles des fonctions et des variables globales, etc dans un **espace de nom non-nommé**, aussi avec **namespace**.

*INTERDIT
pour le projet!!*

4) Enfin, l'**espace de nom global** est celui qui contient les variables globales à l'échelle du programme complet.

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

```
// jamais dans l'interface ! modul.h
```

```
#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){
        ...
    }
}
modul.cc
```

*ici viendrait le nom de l'espace de noms
MAIS comme il n'y a rien, cet
espace de noms est appelé
l'espace de noms non-nommé*

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

```
modul.h
```

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

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

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 connaît 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 connaît les types de l'ensemble des attributs.

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

Type concret \Leftrightarrow 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.

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

int main(void)
{
    S2d centre={0.,0.};
    {
        centre.x = 1.;
        centre.y = 1.;
    }
    ...
}
```

prog.cc

```
struct S2d
{
    double x;
    double y;
};
+ autres struct éventuels
+ prototypes de fonctions
```

tools.h

```
#include <tools.h>

// Définition des fonctions
```

tools.cc

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.

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

int main(void)
{
    Date d;
    d.day   = 31;
    d.month = 2;
    d.year  = 2019;
    ...
}
```

prog.cc

```
struct Date
{
    int day;
    int month;
    int year;
};
+ prototypes de fonctions
```

date.h

```
#include "date.h"

// Définition des fonctions
```

date.cc

Quizz8:

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

```
class Datum{                                     datum.h
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;
};
```

```
#include <iostream>                               datum.cc
#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 correspondance 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 **namespace** ou **static**
- un type *concret* défini avec **struct** doit être réservé pour des entités de bas-niveau avec faible risque d'erreur.