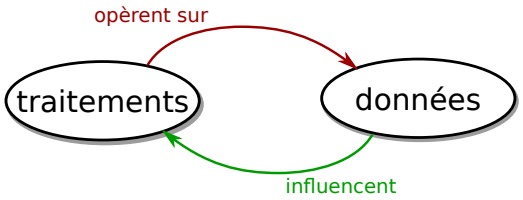
Programmation impérative/procédurale (rappel)

Dans les programmes que vous avez écrits jusqu'à maintenant, les notions

- de variables/types de **données**
- et de **traitement** de ces données

étaient séparées :



Programmation procédurale : exemple

```
double surface(double largeur, double hauteur);

int main()
{
  double largeur(3.0);
  double hauteur(4.0);

  cout << "La surface du rectangle est : "
       << surface(largeur, hauteur) << endl;

  return 0;
}

double surface(double largeur, double hauteur)
{
  return largeur * hauteur;
}
```

Programmation procédurale : exemple

```
struct Rectangle {
  double largeur;
  double hauteur;
};
double surface(Rectangle const& rect);

int main()
{
  Rectangle rectangle({ 3.0, 4.0 });

  cout << "La surface du rectangle est : "
       << surface(rectangle) << endl;

  return 0;
}
double surface(Rectangle const& rect)
{
  return rect.largeur * rect.hauteur;
}
```

Objets : quatre concepts de base

Un des objectifs principaux de la notion d'**objet** :

organiser des programmes complexes

grâce aux notions :

- d'encapsulation
- d'abstraction
- d'héritage
- et de polymorphisme

Notions d'encapsulation

Principe d'encapsulation :
regrouper dans le même objet informatique («concept»), les données et les traitements qui lui sont *spécifiques* :

- **attributs** : les données incluses dans un objet
- **méthodes** : les fonctions (= traitements) définies dans un objet

☞ Les objets sont définis par leurs attributs et leurs méthodes.

Notion d'abstraction

Pour être véritablement intéressant, un objet doit permettre un certain degré d'**abstraction**.

Le processus d'abstraction consiste à identifier pour un ensemble d'éléments :

- des caractéristiques communes à tous les éléments
- des mécanismes communs à tous les éléments

☞ description **générique** de l'ensemble considéré :
se focaliser sur l'essentiel, cacher les détails.

Notion d'abstraction : exemple

Exemple : Rectangles

- la notion d'« *objet rectangle* » n'est intéressante que si l'on peut lui associer des propriétés et/ou mécanismes *généraux* (valables pour l'ensemble des rectangles)
- Les notions de *largeur* et *hauteur* sont des propriétés générales des rectangles (**attributs**),
- Le mécanisme permettant de calculer la surface d'un rectangle (surface = largeur $\times$ hauteur) est commun à tous les rectangles (**méthodes**)

Abstraction et Encapsulation

En plus du regroupement des données et des traitements relatifs à une entité, l'encapsulation permet en effet de définir **deux niveaux** de perception des objets :

- niveau *externe* : partie « *visible* » (par les programmeurs-utilisateurs) :
  - l'**interface** : *prototypes* de quelques méthodes bien choisies
  - ☞ résultat du processus d'*abstraction*
- niveau *interne* : (détails d')**implémentation**
  - **corps** :
    - méthodes et attributs accessibles uniquement depuis l'intérieur de l'objet (ou d'objets similaires)
    - définition de toutes les méthodes de l'objet

Exemple d'interface

L'interface d'une voiture

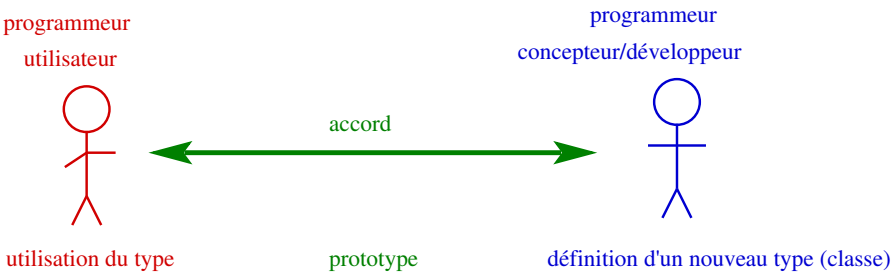
- Volant, accélérateur, pédale de frein, etc.
- Tout ce qu'il faut savoir pour la conduire
  (mais pas la réparer ! ni comprendre comment ça marche)
- L'interface ne change pas, même si l'on change de moteur...
  ...et même si on change de voiture (dans une certaine mesure) :
  abstraction de la notion de voiture (en tant qu'« objet à conduire »)

Encapsulation et Interface

Il y a donc deux facettes à l'encapsulation :

1. regroupement de tout ce qui caractérise l'objet :
   données (attributs) **et** traitements (méthodes)
2. isolement et dissimulation des détails
   d'implémentation
   Interface = ce que le programmeur-utilisateur (hors de l'objet) peut utiliser
   - ☞ Concentration sur les attributs et les méthodes concernant l'objet (*abstraction*)

Les « 3 facettes » d'une classe



Pourquoi abstraire/encapsuler ?

1. L'intérêt de regrouper les traitements et les données conceptuellement reliées est de permettre une *meilleure visibilité* et une meilleure cohérence au programme, d'offrir une plus grande modularité.

```
double largeur(3.0);
double hauteur(4.0);

cout << "Surface : "
     << surface(largeur, hauteur))
     << endl;
```

```
Rectangle rect(3.0, 4.0);
cout << "Surface : "
     << rect.surface()
     << endl;
```

Pourquoi abstraire/encapsuler ? (2)

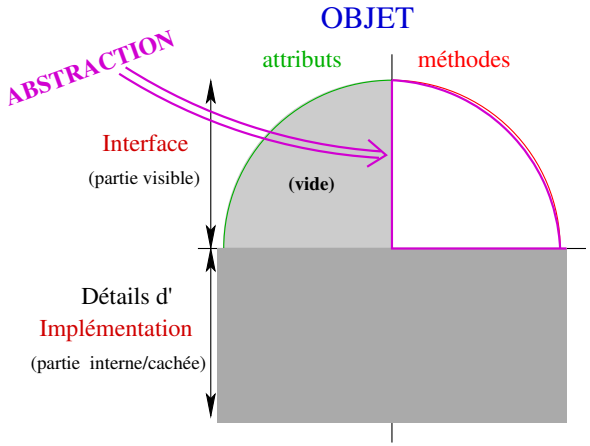
2. L'intérêt de séparer les niveaux *interne* et *externe* est de donner un **cadre** plus **rigoureux** à l'utilisation des objets utilisés dans un programme

   Les objets ne peuvent être utilisés qu'au travers de leurs interfaces (niveau externe)
   et donc les éventuelles **modifications** de la structure interne restent **invisibles** à l'extérieur

   (même idée que la séparation prototype/définition d'une fonction)

Règle : les attributs d'un objet ne doivent pas être accessibles depuis l'extérieur, mais uniquement par des méthodes.

Encapsulation / Abstraction : Résumé

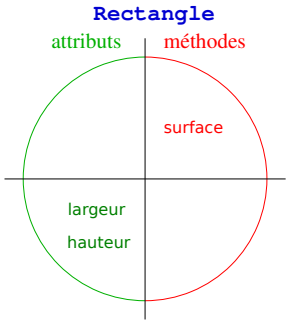


Classes et Instances, Types et Variables

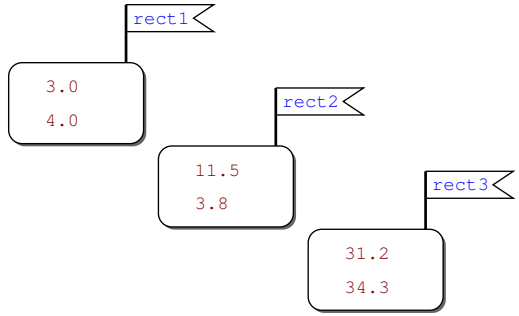
En programmation Objet :

- le résultat des processus d'encapsulation et d'abstraction s'appelle une **classe**
  classe = catégorie d'objets
- une classe définit un **type**
- une réalisation particulière d'une classe s'appelle une **instance**
  instance = **objet**
- un objet est une **variable**

Classes et Instances, Types et Variables (illustration)



classe
type (abstraction)
existence conceptuelle
(écriture du programme)

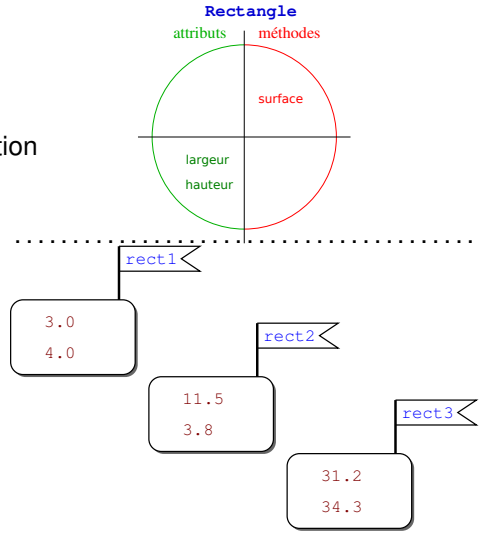


objets/instances
variables en mémoire
existence concrète
(exécution du programme)

Classes et Instances, Types et Variables

En programmation Objet :

- le résultat des processus d'encapsulation et d'abstraction s'appelle une **classe**
  classe = catégorie d'objets
- une classe définit un **type**
- une réalisation particulière d'une classe s'appelle une **instance**
  instance = **objet**
- un objet est une **variable**



Les classes en C++

En C++ une **classe** se déclare par le mot-clé `class`.

Exemple : `class Rectangle { ... };`

Ceci définit un nouveau **type** du langage.

La déclaration d'une **instance** d'une classe se fait de façon similaire à la déclaration d'une **variable** :

nom_classe nom_instance ;

Exemple :

`Rectangle rect1;`

déclare une instance `rect1` de la classe `Rectangle`.

Notre programme (1/4)

```
class Rectangle {
};

int main()
{
  Rectangle rect1;

  return 0;
}
```

Déclaration des attributs

La syntaxe de la déclaration des attributs est la même que celle des champs d'une `structure` :

type nom_attribut ;

Exemple :
les attributs `hauteur` et `largeur`, de type `double`, de la classe `Rectangle` pourront être déclarés par :

```
class Rectangle {
    double hauteur;
    double largeur;
};
```

Accès aux attributs

L'accès aux valeurs des attributs d'une instance de nom `nom_instance` se fait comme pour accéder aux champs d'une structure :

`nom_instance.nom_attribut`

Exemple :
la valeur de l'attribut `hauteur` d'une instance `rect1` de la classe `Rectangle` sera référencée par l'expression :

`rect1.hauteur`

Notre programme (2/4)

```
#include <iostream>
using namespace std;

class Rectangle {
  double hauteur;
  double largeur;
};

int main()
{
  Rectangle rect1;

  rect1.hauteur = 3.0;
  rect1.largeur = 4.0;

  cout << "hauteur : " << rect1.hauteur
       << endl;

  return 0;
}
```

Déclaration des méthodes

La syntaxe de la définition des méthodes d'une classe est la syntaxe normale de définition des fonctions :

```
type_retour nom_methode (type_param1 nom_param1, ...)
{
  // corps de la méthode
  ...
}
```

mais elles sont simplement mises *dans* la classe elle-même.

Exemple : la méthode `surface()` de la classe `Rectangle` :

```
class Rectangle {
  //...
  double surface()
  {
    return hauteur * largeur;
  }
};
```

mais où sont passés les paramètres ?

```
double surface(double hauteur, double largeur)
{
  return hauteur * largeur;
}
```

Portée des attributs

Les attributs d'une classe constituent des variables *directement accessibles* dans toutes les méthodes de la classe (*i.e.* des « variables *globales à la classe* »).

On parle de « *portée de classe* ».

Il n'est donc **pas nécessaire de les passer comme arguments des méthodes**.

Par exemple, dans toutes les méthodes de la classe `Rectangle`, l'identificateur `hauteur` (resp. `largeur`) fait *a priori* référence à la valeur de l'attribut `hauteur` (resp. `largeur`) de l'instance concernée (par l'appel de la méthode en question)

Déclaration des méthodes

Les méthodes sont donc :

- des fonctions propres à la classe
- qui ont donc accès aux attributs de la classe

☞ Il ne faut donc **pas** passer les attributs comme arguments aux méthodes de la classe !

Exemple :

```
class Rectangle {
  //...
  double surface()
  {
    return hauteur * largeur;
  }
};
```

Paramètres des méthodes

Mais ce n'est pas parce qu'on n'a pas besoin de passer les attributs de la classe comme arguments aux méthodes de cette classe, que les méthodes n'ont *jamais* de paramètres.

Les méthodes **peuvent avoir des paramètres** : ceux qui sont nécessaires (et donc *extérieurs à l'instance*) pour exécuter la méthode en question !

Exemple :

```
class FigureColoree {
  // ...
  void colorie(Couleur c)  { /* ... */ }
  // ...
};

FigureColoree une_figure;
Couleur rouge;
// ...
une_figure.colorie(rouge);
// ...
```

Définition externe des méthodes

Il est possible d'écrire les définitions des méthodes à l'extérieur de la déclaration de la classe

☞ **meilleure lisibilité du code**, **modularisation**

Pour relier la définition d'une méthode à la classe pour laquelle elle est définie, il suffit d'utiliser l'*opérateur* `::` *de résolution de portée* :

- La déclaration de la classe contient les *prototypes des méthodes*
- les définitions correspondantes spécifiées à l'extérieur de la déclaration de la classe se font sous la forme :

```
typeRetour NomClasse::nomFonction(type1 param1
                                  , type2 param2
                                  , ...)
{ ... }
```

Définition externe des méthodes : exemple

```
class Rectangle {
  // ...
  double surface(); // prototype
};

// définition
double Rectangle::surface()
{
  return hauteur * largeur;
}
```

Actions et Prédicats

En C++, on peut distinguer les méthodes qui *modifient* l'état de *l'objet* (« **actions** ») de celles qui *ne changent rien* à l'objet (« **prédicats** »).

On peut pour cela ajouter le mot `const` **après** la liste des paramètres de la méthode :

type_retour nom_methode (type_param1 nom_param1, ...) `const`

Exemple :

```
class Rectangle {
  // ...
  double surface() const;
};

double Rectangle::surface() const
{
    return hauteur * largeur;
}
```

Actions et Prédicats

En C++, on peut distinguer les méthodes qui *modifient* l'état de *l'objet* (« **actions** ») de celles qui *ne changent rien* à l'objet (« **prédicats** »).

On peut pour cela ajouter le mot `const` **après** la liste des paramètres de la méthode :

type_retour nom_methode (type_param1 nom_param1, ...) `const`

Si vous déclarez une action en tant que prédicat (`const`), vous aurez à la compilation le message d'erreur :

```
assignment of data-member '...' in read-only structure
```

Appels aux méthodes

L'appel aux méthodes définies pour une instance de nom `nom_instance` se fait à l'aide d'expressions de la forme :

nom_instance.nom_methode(val_arg1, ...)

Exemple : la méthode

```
void surface() const;
```

définie pour la classe `Rectangle` peut être appelée pour une instance `rect1` de cette classe par :

```
rect1.surface()
```

Autres exemples :

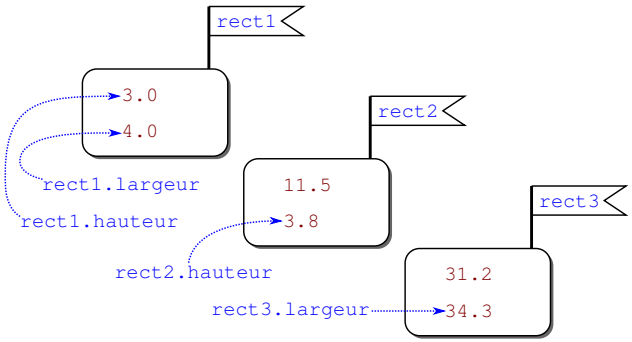
```
une_figure.colorie(rouge);
i < tableau.size()
```

Notre programme (3/4)

```
// ...
class Rectangle {
  double hauteur;
  double largeur;
  double surface() const
    { return hauteur * largeur; }
};

int main()
{
  Rectangle rect1;

  rect1.hauteur = 3.0;
  rect1.largeur = 4.0;

  cout << "surface : " << rect1.surface()
       << endl;
  // ...
```

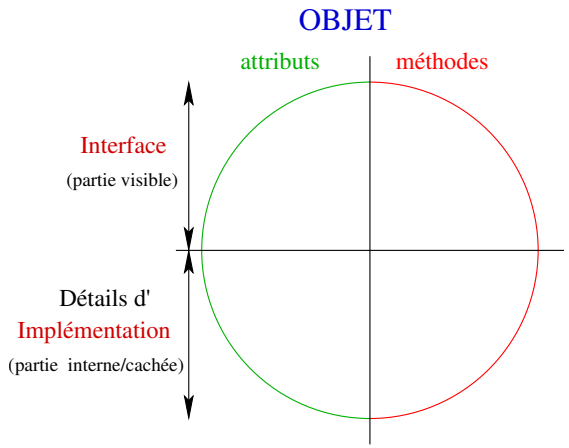
Résumé : Accès aux attributs et méthodes

Chaque instance a ses propres attributs : aucun risque de confusion d'une instance à une autre.



`rect1.surface()` : méthode `surface` de la classe `Rectangle` s'appliquant à `rect1`
`rect1.surface()` ≃ `Rectangle::surface(&rect1)`

Encapsulation / Abstraction



Encapsulation et interface

Tout ce qu'il n'est pas nécessaire de connaître à l'extérieur d'un objet devrait être dans le corps de l'objet et identifié par le mot clé `private` :

```
class Rectangle {
  double surface() const;
private:
  double hauteur;
  double largeur;
};
```

Attribut d'instance **privée** = inaccessible depuis l'extérieur de la classe.
C'est également *valable pour les méthodes*.

Erreur de compilation si référence à un(e) attribut/méthode d'instance privée :

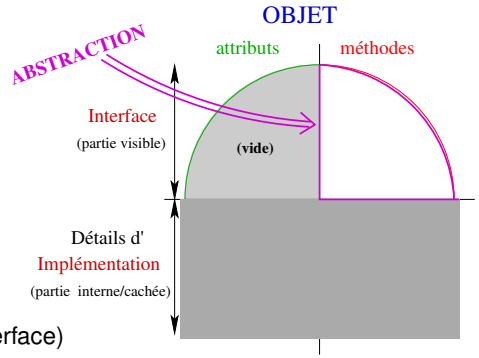
```
Rectangle.cc:16: 'double Rectangle::hauteur' is private
```

Note : Si aucun droit d'accès n'est précisé, c'est `private` par défaut.

Encapsulation et interface (2)

À l'inverse, l'interface, qui est accessible de l'extérieur, se déclare avec le mot-clé `public`

```
class Rectangle {
public:
  double surface() const;
private:
  // ...
};
```



Dans la plupart des cas :

- **Privé** :
  - Tous les attributs
  - La plupart des méthodes
- **Public** :
  - Quelques méthodes bien choisies (interface)

« Accesseurs » et « manipulateurs »

Tous les attributs sont privés ?

- Et si on a besoin de les utiliser depuis l'extérieur de la classe ? !

  Par exemple, comment « manipuler » la largeur et la hauteur d'un rectangle ?

« Accesseurs » et « manipulateurs »

Si le programmeur *le juge utile*, il **inclut les méthodes publiques nécessaires** ...

1. Accesseurs (« méthodes get » ou « getters ») :
   - Consultation (i.e. « prédicat »)
   - Retour de la valeur d'une variable d'instance précise

```
double getHauteur() const { return hauteur; }
double getLargeur() const { return largeur; }
```

2. Manipulateurs (« méthodes set » ou « setters ») :
   - Modification (i.e. « action »)
   - Affectation de l'argument à une variable d'instance précise

```
void setHauteur(double h) { hauteur = h; }
void setLargeur(double l) { largeur = l; }
```

Notre programme (4/4)

```
#include <iostream>
using namespace std;

class Rectangle {
public:
  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:
  double hauteur;
  double largeur;
};

int main()
{
  Rectangle rect1;

  rect1.setHauteur(3.0);
  rect1.setLargeur(4.0);

  cout << "hauteur : "
       << rect1.getHauteur()
       << endl;

  return 0;
}
```

« Accesseurs », « manipulateurs » et encapsulation

Fastidieux d'écrire des « Accesseurs »/« manipulateurs » alors que l'on pourrait tout laisser en `public` ?

```
class Rectangle
{
public:
  double largeur;
  double hauteur;
  string label;
};
```

mais dans ce cas ...

```
Rectangle rect;
rect.hauteur = -36;
cout << rect.label.size() << endl;
```

Masquage (shadowing)

masquage = un identificateur « cache » un autre identificateur

```
int main() {
  int i(120);

  for (int i(1); i < MAX; ++i) {
    cout << i << endl;
  }

  cout << i << endl;
  return 0;
}
```

Situation typique en POO : un paramètre cache un attribut

```
void setHauteur(double hauteur) {
  hauteur = hauteur;  // Hmm.... pas terrible !
}
```

Masquage et pointeur `this`

Si, dans une méthode, un attribut est **masqué** alors la valeur de l'attribut peut quand même être référencée à l'aide du mot réservé `this`.

`this` est un **pointeur sur l'instance courante**

`this` $\simeq$ « mon adresse »

Syntaxe pour spécifier un attribut *en cas d'ambiguïté* :
`this->nom_attribut`

Exemple :

```
void setHauteur(double hauteur) {
  this->hauteur = hauteur;  // Ah, là ça marche !
}
```

L'utilisation de `this` est obligatoire dans les situations de **masquage** (mais évitez ces situations !)

Portée des attributs (résumé)

La portée des attributs dans la définition des méthodes est résumée par le schéma suivant :

```
class MaClasse {
private:
  int x;
  int y;
public:
  void une_methode( int x ) {
    ... y ...
    ... x ...
    ... this->x ...
  }
};
```

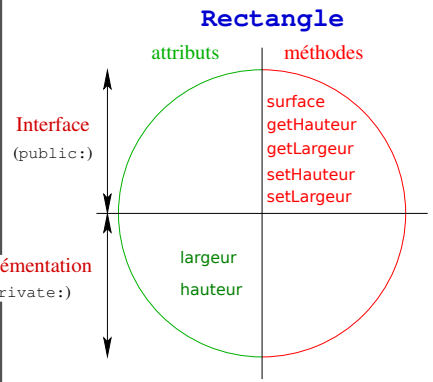
Classes = super `struct`

Pour résumer à ce stade, une classe est une `struct`

- qui contient aussi des fonctions (« méthodes »)
- dont certains champs (internes) peuvent être cachés (`private:`)
- et dont d'autres constituent l'interface (`public:`)

Un exemple complet de classe (1/2)

```
#include <iostream>
using namespace std;

// 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;
};
```



Un exemple complet de classe (2/2)

```
//utilisation de la classe

int main()
{
  Rectangle rect;
  double lu;
  cout << "Quelle hauteur ? "; cin >> lu;
  rect.setHauteur(lu);
  cout << "Quelle largeur ? "; cin >> lu;
  rect.setLargeur(lu);

  cout << "surface = " << rect.surface()
       << endl;

  return 0;
}
```

Exemple avec définitions externes à la classe

```
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_;
};
```

```
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
```

Exemple : jeu de Morpion (1)

On veut coder une classe permettant de représenter le plateau 3x3 d'un jeu de Morpion (tic-tac-toe) :

```
typedef array<int, 9> Grille;

class JeuMorpion {
public:
  void initialise() { grille = new Grille; }
  Grille* get_grille() { return grille ; }

private:
  Grille* grille;
};
```

Exemple : jeu de Morpion (2)

```
typedef array<int, 9> Grille;

class JeuMorpion {
public:
  void initialise() { grille = new Grille; }
  Grille* get_grille() { return grille ; }

private:
  Grille* grille;
};
```

Le joueur rond coche la case en haut à gauche :

```
JeuMorpion jeu;
jeu.initialise();
(*jeu.get_grille())[0] = 1;
```

Convention : 1 représente un rond, 2 une croix et 0 une case vide

Exemple : jeu de Morpion (3)

Ce code est parfaitement fonctionnel mais ... pose **beaucoup** de problèmes :

- L'utilisateur de la classe `JeuMorpion` doit savoir que les cases sont stockées sous forme d'entiers dans un tableau 1D, ligne par ligne (et non colonne par colonne)
- Il doit savoir que la valeur entière 0 correspond à une case non cochée, que 1 correspond à un rond, et que la valeur 2 correspond à une croix.

☞ L'utilisateur doit connaître « le codage » des données

```
JeuMorpion jeu;
jeu.initialise();
(*jeu.get_grille())[0] = 1;
```

Exemple : jeu de Morpion (4)

```
JeuMorpion jeu;
jeu.initialise();
(*jeu.get_grille())[0] = 1;
```

- Le code est complètement cryptique pour une personne qui n'est pas intime avec les entrailles du programme. 0, 1, 2 ? Que cela signifie-t-il ? Impossible de le deviner juste en lisant ce code. Il faut aller lire le code de la classe `JeuMorpion` (ce qui devrait être inutile), et en plus ici `JeuMorpion` n'est même pas documentée !
- Le code n'est pas encapsulé : on a un accesseur public vers une variable privée, donc... on ne peut pas la modifier, non ? Malheureusement si : c'est un pointeur, donc on peut directement modifier ce qu'il pointe ce qui viole l'encapsulation.
- Que se passerait-il si pour représenter le plateau de jeu, on décidait de changer et d'utiliser un tableau 2D ? Ou 9 variables entières ?
  - ☞ Le code écrit par l'utilisateur de la classe `JeuMorpion` serait à réécrire !

Exemple : jeu de Morpion (5)

- Si l'utilisateur s'avisait de faire `(*jeu.get_grille())[23] = 1` ; il aurait un message d'erreur (un segmentation fault)
- Si l'utilisateur avait envie de mettre la valeur 3 ou 11 ou 42 dans le tableau, rien ne l'en empêche - mais d'autres méthodes, comme par exemple `get_joueur_gagnant()`, qui s'attendent uniquement aux valeurs 1 et 2 ne fonctionneront plus du tout !
- Si l'utilisateur avait envie de tricher et de remplacer un rond par une croix ? Il suffit d'écraser la valeur de la case avec la valeur 2 !

☞ Les méthodes choisies ici donnent un accès non contrôlé aux données et n'effectuent aucune **validation** des données

Jeu de Morpion : bien encapsuler (1)

```
enum CouleurCase { VIDE, ROND, CROIX };
typedef array<array<CouleurCase, 3>, 3> Grille;

class JeuMorpion {
public:
  void initialise() {
    for (auto& ligne : grille) {
      for (auto& kase : ligne) {
        kase = VIDE;
      }
    }
  }
private:
  Grille grille;
  //...
```

Jeu de Morpion : bien encapsuler (2)

```
private:
/**
 * Place un coup sur le plateau en position (ligne, colonne).
 * Ligne et colonne : 0, 1 ou 2.
 */
bool placer_coup(size_t ligne, size_t colonne, CouleurCase coup) {
  if ( ligne >= grille.size()
       or colonne >= grille[ligne].size() ) {
    // traitement de l'erreur ici
  }
  if (grille[ligne][colonne] == VIDE) {
    // case vide, on peut placer le coup
    grille[ligne][colonne] = coup;
    return true;
  } else {
    // case déjà prise, on signale une erreur.
    // ...
    return false;
  } // suite
```

Jeu de Morpion : bien encapsuler (3)

```
public:
bool placer_rond( size_t ligne, size_t colonne) {
  return placer_coup(ligne, colonne, ROND);
}
bool placer_croix(size_t ligne, size_t colonne) {
  return placer_coup(ligne, colonne, CROIX);
}

// ici on peut rajouter une methode get_joueur_gagnant() const

}; // fin de la classe JeuMorpion
```

Jeu de Morpion : bien encapsuler (4)

Comment faire maintenant pour faire un rond sur la case du milieu ?

```
JeuMorpion jeu;
jeu.initialise();
valide = jeu.placer_rond(1, 1); // bool valide déclaré plus haut
```

Et pour faire une croix sur la 1re ligne, 2e colonne ?

```
valide = jeu.placer_croix(0, 1);
```

On aurait pu également décider d'appeler les colonnes 1, 2, 3 au lieu de 0, 1, 2 : c'est une question de convention. C'est justement ce sur quoi il faut se mettre d'accord quand on définit une interface.

Jeu de Morpion encapsulé : avantages

- **Validation** : il est impossible de mettre une valeur invalide dans le tableau (autre que 0, 1, ou 2)
- **Validation** : il est impossible de cocher une case déjà cochée.
- **Séparation des soucis** : le programmeur-utilisateur n'a pas besoin de savoir comment le plateau est stocké, ni qu'il utilise des entiers, ni quelles valeurs correspondent à quoi.
- Le code est compréhensible même par un profane – le nom des méthodes exprime clairement ce qu'elles font et s'explique de lui-même.
- Si on essaie de faire une opération invalide (cocher deux fois la même case, ou une case en dehors du tableau), on obtient un message compréhensible.