

Support de cours

Cours:

Introduction à la programmation orientée objet (en C++)

Vidéo:

W11-05-morpions-CPP-pt3

Concepts (extraits des sous-titres générés automatiquement) :

Implémentation de la grille de jeu. Classe jeumorpion. Utilisateur de la grille. Case de la grille. Grille de jeu. Présence d'autres méthodes. Nombre de méthodes. Utilisateur de la classe jeumorpion. Logique du jeu. Programmeur externe. Alignements de ronds. Version précédente. Concepteur de la classe jeumorpion. Programmeur-concepteur. Implémentation d'autres classes.



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en C++).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>

Encapsulation et abstraction : étude de cas

(Partie 3)

Introduction à la programmation orientée objet (en C++)

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



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

Il résulte de l'accès à l'implémentation de la grille de jeu via le pointeur toutes sortes de conséquences fâcheuses. Par exemple, rien n'empêche ce genre de manipulations : l'accès à une case de la grille via un indice qui est invalide. Celui qui utilise `get_grille` ici peut s'être trompé dans le calcul de l'indice, donc l'erreur peut ne pas être aussi explicite qu'elle l'est ici, mais rien n'empêche de faire un accès de cette nature. A ce moment-là, le programme aura évidemment un comportement erroné qui peut, dans certaines situations, se traduire par des erreurs telles que des segmentation fault et des arrêts abrupts du programme. Rien n'empêche non plus de mettre dans la grille de jeu, des valeurs autres que celles qui sont initialement prévues, à savoir des ronds ou des croix ce qui correspond à "1" ou "2". On pourrait par exemple imaginer que par inadvertance ou par malice, celui qui utilise la grille de jeu y injecte des valeurs qui ne soient pas celles attendues. Des valeurs fantaisistes comme 42, 11 ou 3. A ce moment-là, on se trouverait dans une situation où la logique du jeu ne serait plus respectée. Imaginez par exemple que celui qui a programmé la logique du jeu, ait programmé une méthode `get_joueur_gagnant` dont le rôle est de parcourir la grille de jeu pour vérifier s'il y a des alignements de ronds ou de croix. A ce moment-là, cette méthode s'attend évidemment à trouver des valeurs telles que 1 et 2 et aurait un comportement erroné si elle se trouvait en face d'une valeur telle que 42.

notes

résumé

0m 1s





Enfin, imaginez le cas où l'utilisateur de la grille aurait envie de tricher, c'est-à-dire de remplacer un rond par une croix qui serait déjà placée sur la grille de jeu, rien ne l'en empêche. Donc ici on peut tout à fait, après avoir déjà mis un "1" ici, remplacer ce "1" par un "2" en utilisant ce genre de manipulations. Donc aucun contrôle n'est fait à ce niveau pour vérifier qu'il n'y a pas déjà un pion placé sur la grille de jeu et que l'écrasement n'est pas possible. Tous les problèmes que l'on a évoqués ici résultent du fait que les fonctionnalités que l'on a choisi d'offrir publiquement sont de trop bas niveau, révèlent trop les détails d'implémentation et nécessitent de les connaître. Elles offrent par conséquent un accès incontrôlé aux données et permettent d'utiliser ces données sans aucune validation, ce qui bien évidemment correspond à une mauvaise conception.

notes

résumé

1m 25s



```
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;
    //...
```

Pour programmer une classe JeuMorpion proprement encapsulée, il est impératif que les choix d'implémentations ne soient pas accessibles, visibles au travers de l'interface d'utilisation qui est offerte au programmeur externe. Par ailleurs, les choix d'implémentations internes, comme par exemple comment représenter la grille, comment modéliser les jetons qui se posent sur la grille ; doit être du ressort de celui qui implémente la classe JeuMorpion et non pas de celui qui l'utilise. Nous allons étudier dans ce qui suit, une variante qui améliore la version précédente. Dans cette seconde version de la classe JeuMorpion, le programmeur-concepteur décide comme il se doit,

notes

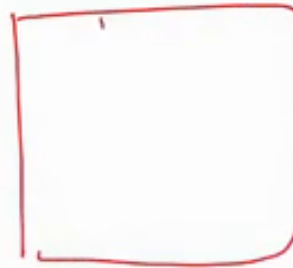
résumé

2m 16s



```
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;
    //...
```



de prendre en charge la modélisation de tous les éléments permettant de modéliser un jeu de morpion. C'est à dire non seulement la grille de jeu, mais également les pions que l'on va y placer. Et il choisit pour cela d'introduire un type énuméré qui va permettre de manipuler ces jetons via des noms qui sont explicites et parlants. Pour la modélisation de la grille de jeu elle-même, le programmeur fait des choix qui sont plus naturels que ceux que l'on a vu précédemment. Il va utiliser un tableau à 3 lignes et 3 colonnes, ce qui est beaucoup plus proche de la représentation que l'on a

notes

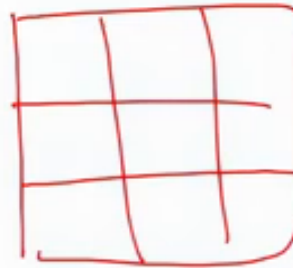
résumé

2m 49s



```
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;
    //...
```



d'un plateau du jeu de morpion. Dans la classe JeuMorpion,

notes

résumé

3m 24s



```
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
```

il n'y a pas de nécessité d'accéder à la grille de jeu via un pointeur. L'attribut est donc naturellement défini comme étant de type grille tel que défini ici. De plus, l'initialisation de la grille qui n'était que partielle et mal faite dans la version précédente, est beaucoup plus correcte ici. Le concepteur de la classe JeuMorpion prend en charge l'initialisation de tous les éléments de la grille, c'est-à-dire également de son contenu. Et donc pour ce faire, il va balayer chacune des lignes de la grille et pour chaque case de cette ligne, en remplir le contenu en y mettant la valeur "vide" de façon totalement explicite. L'idée est donc maintenant d'offrir à l'utilisateur de la classe JeuMorpion, un certain nombre de méthodes bien choisies qui vont lui permettre de manipuler la grille de jeu sans en connaître les détails d'implémentation. On peut donc par exemple imaginer d'offrir une méthode `placer_rond` dont le rôle sera de placer un rond sur la grille de jeu à une ligne et une colonne fournies par l'utilisateur. De même, on aurait le tenant `placer_croix` qui placerait une croix sur une ligne et une colonne données. Ces méthodes retourneraient par exemple un booléen indiquant si le placement du rond ou de la croix a pu se faire à la ligne et à la colonne souhaitées. On peut bien sûr imaginer la présence d'autres méthodes, comme par exemple une méthode `get_joueur_gagnant` qui analyserait le contenu de la grille pour savoir s'il y a un alignement de ronds ou de croix. Le fait de placer un jeton à une ligne et une colonne données se fait en fait exactement selon les mêmes modalités, peu importe que l'on place un rond ou une croix. Il est donc naturel ici de penser à modulariser les traitements et de créer une méthode `placer_coup` qui va se charger de placer les différentes pièces sur la grille de jeu. Les deux

notes

résumé

3m 29s




```
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
```

méthodes `placer_rond` et `placer_croix` suffisent complètement à l'utilisateur de la classe `JeuMorpion` pour pouvoir placer un jeton sur la grille de jeu. Il n'est par conséquent pas nécessaire d'offrir également l'accès à `placer_coup` qui ici peut être considéré comme une méthode utilitaire aidant `placer_croix` et `placer_rond` à faire leur travail mais qui ne fera pas partie de l'interface d'utilisation de la classe. Les deux méthodes `placer_rond` et `placer_croix` suffisent complètement à l'utilisateur de la classe `JeuMorpion` pour pouvoir placer un jeton sur la grille de jeu. Il n'est par conséquent pas nécessaire d'offrir également l'accès à `placer_coup`

notes

résumé

```
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
}
```

qui ici peut être considéré comme une méthode utilitaire aidant placer_croix et placer_rond à faire leur travail mais qui ne fera pas partie de l'interface d'utilisation de la classe. Par conséquent, on va décider de déclarer cette méthode placer_coup comme privée. La méthode placer_coup est donc en charge de placer un coup qui peut être un rond ou une croix à une ligne et une colonne données. Comme ces informations ligne et colonne sont fournies par celui qui utilise la classe au travers de placer_rond et placer_croix, il faut faire un travail de validation pour vérifier que les lignes et les colonnes souhaitées pour l'introduction du coup correspondent bien à des lignes et des colonnes valides. La méthode placer_coup commence donc par faire cette vérification. Si cette vérification échoue, on doit traiter l'erreur et donc indiquer par un return_false que le placement n'a pas pu se faire correctement. La méthode placer_coup doit également vérifier que l'on ne place pas un jeton sur une case qui est déjà occupée, ce qui est contraire à la logique du jeu. Et donc on va faire cette vérification explicitement en testant si la position souhaitée ne correspond pas à une case qui est vide. Alors dans ce cas-là, la case est déjà prise, on signale une erreur et on retourne fault. Dans l'autre cas, on va placer notre coup à la position souhaitée et on va retourner true pour indiquer que placer_coup s'est correctement passé. Vous noterez qu'il n'a pas été nécessaire ici de vérifier la validité du coup,

notes

résumé

5m 49s



```
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
```

de savoir si on a placé un rond ou une croix parce que cette méthode `placer_coup` est une méthode privée qui n'est invoquée que par les méthodes `placer_rond`

notes

résumé

7m 13s



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 1^{re} ligne, 2^e colonne ?

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

et placer_croix qui explicitement mettent déjà les bonnes valeurs pour le coup à placer. Voyons maintenant comment toutes ces modifications de la classe JeuMorpion se répercutent sur celui qui l'utilise. Donc quelqu'un qui utiliserait la classe commencerait vraisemblablement par déclarer une variable de type JeuMorpion. Ensuite, il utiliserait la méthode publique offerte pour l'initialisation afin de partir avec une grille de jeu vide. Et ensuite il commencerait à s'intéresser à placer différents jetons, différentes pièces, sur la grille de jeu. Si l'utilisateur souhaite par exemple placer un rond à la ligne 1 et à la colonne 1 de la grille de jeu, il va pouvoir utiliser une tournure de cette nature qui est beaucoup plus claire et beaucoup plus explicite que celle que nous utilisions précédemment. Il indique simplement qu'il place un rond sur le jeu à la ligne 1 et à la colonne 1. Surtout, cette tournure ne nécessite de sa part aucune connaissance sur les détails d'implémentation de la classe JeuMorpion. L'utilisateur peut aussi bénéficier du travail de validation des données qui est faite par des méthodes telles que placer_rond ou placer_croix. Donc il peut s'informer par le biais du booléen qui est livré par ces méthodes et savoir si le placement de son coup s'est passé favorablement ou pas. Le placement d'une croix sur la grille de jeu va se faire évidemment selon les mêmes modalités. La nouvelle version de la classe JeuMorpion nous permet une utilisation beaucoup plus robuste et beaucoup plus explicite.

notes

résumé

7m 23s



- **Validation** : il est impossible de mettre une valeur invalide dans le tableau (autre que 0, 1, ou 2)

Elle ne nous affranchit cependant pas complètement de devoir expliciter un certain nombre de conventions d'usages des méthodes de l'interface. Par exemple, il faut informer l'utilisateur du fait que l'on s'attend à une numérotation des lignes et des colonnes selon une convention qui numérote la première ligne à 0 et non pas à 1. C'est ce genre de convention qu'il faut absolument documenter lorsqu'on fournit l'interface d'utilisation. La deuxième version de la classe JeuMorpion offre de nombreux avantages par rapport à la première.

notes

résumé

8m 49s



- ▶ **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 premier avantage est la validation des données. Il devient désormais impossible à l'utilisateur de la classe JeuMorpion de placer une valeur invalide dans le tableau, autre que les valeurs que l'on attend. En fait, il ne sait même pas comment ces valeurs sont codifiées en interne. Il ne sait pas qu'on a par exemple utilisé un type énuméré pour représenter chacune de ces valeurs. Il n'a aucune idée de comment tout ceci est implémenté en interne. Toujours dans le cadre de la validation des données, il est impossible désormais à l'utilisateur de cocher une case déjà cochée. Ce n'était pas le cas dans la version précédente. Ici, les méthodes de l'interface sont en charge de faire ce genre de travail de vérification et interdiront ce genre de manipulations non souhaitées. Autre point très important : cette mise en oeuvre de la classe JeuMorpion permet ce que l'on appelle « la séparation des soucis ». Chaque programmeur est responsable de la partie qui le concerne et n'a pas à se préoccuper des soucis relatifs à l'implémentation d'autres classes. Par exemple, celui qui utilise la classe JeuMorpion, n'a nul besoin de savoir comment le plateau de jeu est stocké, ni qu'il utilise des entiers, ni quelles valeurs correspondent à quoi que ce soit. Celui qui est en charge d'implémenter JeuMorpion se charge de ces détails. Celui qui utilise n'est en charge que de la logique du jeu. Autre point important : la lisibilité du code qui résulte de cette nouvelle implémentation. Le code produit par l'utilisateur de JeuMorpion est désormais compréhensif même pour quelqu'un qui n'a pas de connaissances approfondies. Donc ici, le fait d'écrire quelque chose comme `jeu.placer_rond(...)` est tout à fait compréhensible. Il n'y a pas besoin de beaucoup d'explications pour savoir ce que ce traitement réalise. Ça n'était pas le cas dans l'implémentation précédente où je devais passer par une tournure totalement cryptique pour réaliser un traitement analogue.

notes

résumé

9m 23s



- ▶ **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.

Enfin, conséquence de la validation des données, il devient possible d'informer l'utilisateur en cas de situation anormale en utilisant un message compréhensible. Si maintenant on essaie de placer un jeton dans une position invalide de la grille, c'est le traitement d'erreur réalisé par `placer_coup` qui va informer l'utilisateur que quelque chose d'anormal est en cours et qu'il ne peut pas placer le coup. Dans la version précédente, le programme s'arrêtait avec une erreur de type `segmentation fault`.

notes

résumé