Support de cours

Cours:

Initiation à la programmation (en C++)

Vidéo:

Puissance 4 - premières fonctions

Concepts (extraits des sous-titres générés automatiquement) :

Grille de jeu. Codage d'une fonction. Type de données. Affichage de la grille. Codage d'une seconde fonction. Lignes de codes. Types fondamentaux. Boucle for. Éléments de la grille. Premières fonctionnalités. Codage d'une application. Codage du jeu du puissance. Fonctionnalités fondamentales. Fait naturel. Ensemble de valeurs.

vers la recherche de séquences vidéo
(dans Initiation à la programmation (en C++).)

vers la vidéo

EPFL

Puissance 4 : premières fonctions

Initiation à la programmation (C++)

Vincent Lepetit, Jean-Cédric Chappelier et Jamila Sam

...

notes

résumé

0m 0s

Le codage d'une application, quelle qu'elle soit,

notes

résumé

0m 1s

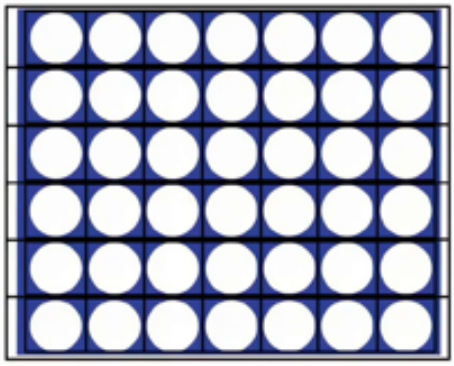
Premières fonctions

EPFL

```
enum Couleur { vide, rouge, jaune };

typedef array<array<Couleur, 7>, 6> Grille;
```

- `initialise` : fonction qui initialise une grille ;
- `affiche` : fonction qui affiche une grille.



nécessite de réfléchir au type de données impliquées ainsi qu'aux fonctionnalités fondamentales qui vont permettre de mettre en œuvre les traitements souhaités. Nous avons entamé cette démarche lors de la séquence précédente pour le codage du jeu du Puissance 4. Ce qui nous a permis d'identifier les types fondamentaux pour représenter la grille de jeu. Nous allons aujourd'hui poursuivre notre réflexion en nous penchant sur les fonctionnalités de bases. Pour rappel, lors de la séquence précédente nous avions réfléchi aux types de données impliquées. Très naturellement, nous avions réfléchi à comment modéliser la grille de jeu. Ceci nous avait conduit à définir un nouveau type, le type « Grille », comme un tableau à deux dimensions. Nous avions également réfléchi à modéliser le contenu de chacune des cellules de chacun des éléments de ce tableau. Ce qui nous avait conduit à introduire un type énuméré, le type « Couleur », qui nous permet justement de représenter le contenu de chacune des cellules du tableau. La grille étant de taille fixe notre choix s'est naturellement porté sur l'utilisation du type « array » de C++2011 qui nous permet effectivement de représenter un tableau de taille fixe. Nous avons simplement déclaré le type « Grille » comme étant un type capable de représenter un tableau à deux dimensions de taille fixe ayant 6 lignes et 7 colonnes. Chacun des éléments de la grille est de type « Couleur », « Couleur » étant le type énuméré permettant les valeurs « vides », « rouge », « jaune ». Une fois ces types introduits je peux tout à fait dans un programme déclarer une variable de type « Grille » et affecter des valeurs à ces différents éléments. Ces valeurs seront prises dans le cadre du type énuméré. c'est à dire pouvant être vide, rouge

notes

résumé

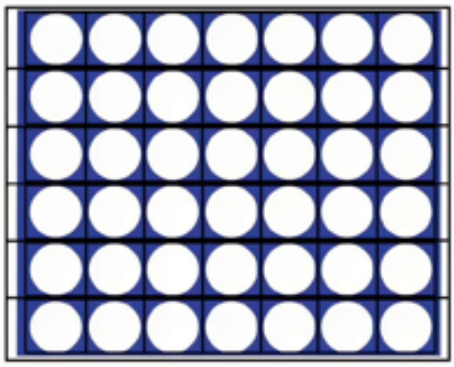
0m 5s

Premières fonctions

```
enum Couleur { vide, rouge, jaune };

typedef array<array<Couleur, 7>, 6> Grille;
```

- `initialise` : fonction qui initialise une grille ;
- `affiche` : fonction qui affiche une grille.



ou jaune. Les types fondamentaux utiles étant définis, nous nous intéressons maintenant aux premières fonctionnalités à mettre en œuvre sur la grille. Il est vraisemblable que lorsque nous commençons à jouer, nous souhaitons commencer à jouer sur une grille qui est vide. Il faudra nous pencher sur l'initialisation permettant d'initialiser chacune des cellules du tableau à la valeur vide. Et bien évidemment, il faudra aussi permettre l'affichage de la grille, afin que les joueurs puissent se rendre compte de la progression du jeu. Nous allons dans le cadre de cette séquence, nous intéresser à l'implémentation concrète de ces deux fonctionnalités. Nous prévoyons le codage d'une fonction « initialise » qui permet l'initialisation de la grille,

notes

résumé

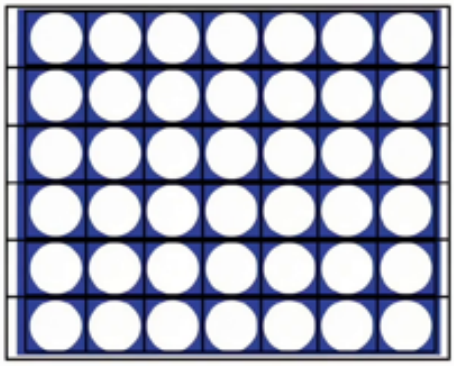
Fonction initialise

```
enum Couleur { vide, rouge, jaune };

typedef array<array<Couleur, 7>, 6> Grille;

...

Grille grille;
```



ainsi que le codage d'une seconde fonction appelée « affiche » qui permet l'affichage de la grille. Commençons par initialisation. Commençons par imaginer comment s'utiliserait concrètement cette fonction « initialise » dans un programme. On peut imaginer que dans un programme j'ai au préalable déclaré

notes

résumé

2m 25s

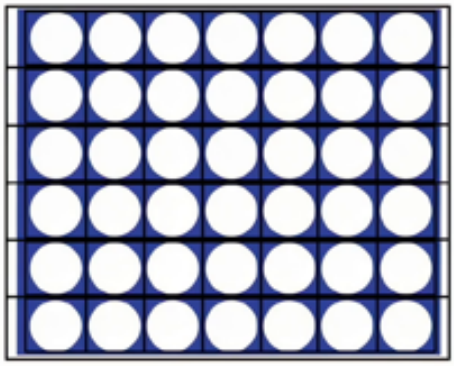
Fonction initialise

```
enum Couleur { vide, rouge, jaune };

typedef array<array<Couleur, 7>, 6> Grille;

...

Grille grille;
initialise(grille);
```



une variable « grille » de type « Grille » et pour initialiser cette grille je peux imaginer que je vais rédiger une fonction qui pourrait être invoquée comme ceci. Donc j'appelle la fonction initialise qui prendrait en paramètre la grille pour en modifier le contenu. Modifier le contenu de quelle façon ? Et bien simplement, en remplissant chacune des cellules avec la valeur « vide ». Intéressons-nous maintenant au codage de cette fonction « initialise ». Je vois que tel que j'ai imaginé de l'invoquer ici, cette fonction initialise n'a pas besoin de me fournir un résultat. Sa vocation est de modifier la grille, de l'initialiser. Donc très naturellement, je vais décrire une fonction « initialise »

notes

résumé

2m 42s

Fonction initialise

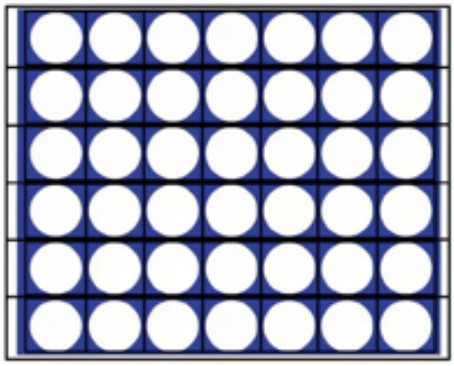
```
enum Couleur { vide, rouge, jaune };

typedef array<array<Couleur, 7>, 6> Grille;

void initialise(Grille& grille)
{
 | for(auto &kase : grille[0]) { // "case" est un mot-cle reserve du C++
    kase = vide;
  }
}

...

Grille grille;
initialise(grille);
```



dont le type de retour est « void » puisque je n'ai pas besoin de récupérer une valeur à la sortie de l'exécution de cette fonction « initialise ». Je rédige ma fonction initialise en lui passant en paramètre une « Grille ». La fonction « initialise » doit modifier la grille. Donc une fois que j'ai terminé l'exécution de ma fonction, à ce stade du programme, je dois récupérer une grille dont le contenu a été modifié. Donc pour cette raison, je suis dans l'obligation d'utiliser un passage par référence afin que la grille sorte bel et bien modifiée de la fonction d'initialisation. Notez que cette façon de concevoir la fonction « initialise » n'est évidemment pas la seule solution possible. On pourrait imaginer ici qu'au lieu que la fonction « initialise » travaille directement sur une grille fournie pour la remplir, on pourrait imaginer que cette fonction « initialise » retourne une grille initialisée de la bonne façon. Ce qui correspondrait à un appel suivant. Ici on remplacerait les deux lignes de codes que l'on a écrites ici par quelque chose qui ressemblerait plutôt à cela. On invoquerait une fonction « initialise » qui au lieu de construire une grille donnée, retourne une grille. A ce moment-là le type de retour ne serait plus « void » mais on retournerait bel et bien une grille. Ici, il n'y aurait plus besoin d'argument, puisque la fonction « initialise » retourne une grille. Donc c'est une autre façon possible que nous avons déjà ébauchée dans notre cours sur les structures, lorsque nous avons pris l'exemple de la fonction qui permettait de faire naître une personne. Aujourd'hui, nous avons choisi une conception alternative pour illustrer notamment l'idée du passage par référence. L'en-tête de la fonction est maintenant clairement défini et

notes

résumé

3m 23s

Fonction initialise

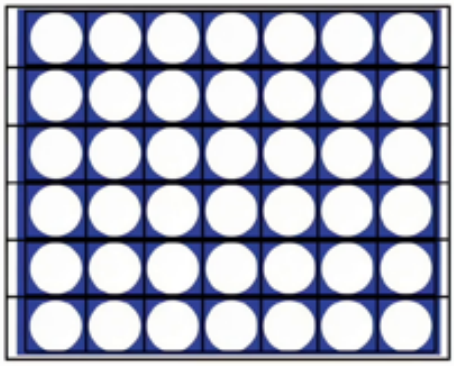
```
enum Couleur { vide, rouge, jaune };

typedef array<array<Couleur, 7>, 6> Grille;

void initialise(Grille& grille)
{
 | for(auto &kase : grille[0]) { // "case" est un mot-cle reserve du C++
    kase = vide;
  }
}

...

Grille grille;
initialise(grille);
```



il s'agit de commencer à développer le corps de la fonction. Intéressons-nous au contenu du corps de la fonction. Ce que doit faire cette fonction c'est pour chacune des lignes du tableau, itérer sur chacune des cellules et y mettre la valeur vide. Commençons par voir comment ce traitement se réalise pour initialiser une seule ligne. Imaginons de le faire pour la première ligne de la grille. Il faudrait évidement écrire une « boucle for ». Cette « boucle for » va itérer sur chacun des éléments de la ligne. Donc on peut imaginer une « itération sur ensemble de valeurs » qui s'écrit de cette façon. Je le fais d'abord pour la première ligne de ma grille, en guise d'introduction ici, pour chacune des cases de la première ligne de la grille. Ce qui se rédige comme ça. Je vais remplir le contenu de sorte à y mettre la valeur vide. A ce stade, en guise d'exercice, j'ai écrit les lignes de codes me permettant d'initialiser la première ligne de la grille en mettant dans chacune de ces cellules la valeur vide. Ici, je peux simplement clore ma fonction. Vous noterez au passage l'utilisation d'un type « référence » pour la variable d'itération dans mon « itération sur ensemble de valeurs ». Ceci est impératif si je veux que le contenu de chaque case soit modifié. Si je compile ce programme en l'état, je vais cependant butter sur un petit problème. L'identificateur « case » très naturel en français pour identifier une case du tableau est en réalité un mot réservé du langage C++. Je ne peux pas l'utiliser comme un identificateur de variable. Dans ce cas-là, je dois apporter une petite rectification pour que ce programme puisse effectivement compiler. Je vais changer le nom

notes

résumé

Fonction initialise

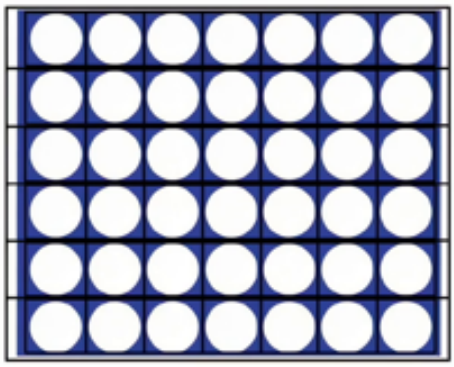
```
enum Couleur { vide, rouge, jaune };

typedef array<array<Couleur, 7>, 6> Grille;

void initialise(Grille& grille)
{
 | for(auto &kase : grille[0]) { // "case" est un mot-cle reserve du C++
    kase = vide;
  }
}

...

Grille grille;
initialise(grille);
```



de cet identificateur. Je peux prendre la précaution de commenter la raison de ce choix. En expliquant que je ne peux pas utiliser le mot réservé « case » puisque c'est un mot réservé du langage. Evidemment, il faudra rectifier l'identificateur là où il est utilisé. Comme ceci. Evidemment, ce traitement que j'ai mis en œuvre pour initialiser la première ligne de ma grille doit être répété pour chacune des lignes afin d'initialiser l'ensemble de la grille. Il est naturel ici d'imaginer d'imbriquer ces traitements dans une première « boucle for » qui elle va être en charge d'itérer sur l'ensemble des lignes et réaliser exactement le même traitement.

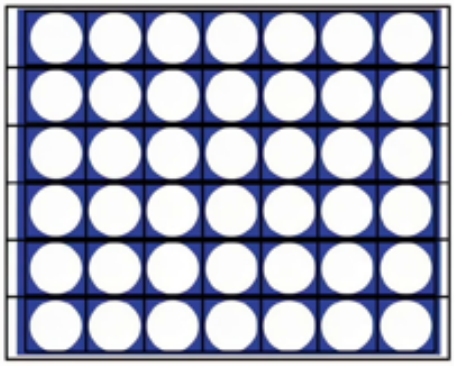
notes

résumé

affiche

```
...

Grille grille;
initialise(grille);
```



Réalisons et écrivons cette première « boucle for », qui va itérer sur l'ensemble des lignes de la grille. A nouveau, je peux utiliser une « itération sur ensemble de valeurs ». Donc avec une variable « ligne » qui me permet d'itérer sur chacune des lignes de ma grille. Pour chacune des lignes, je remplace ici « grille0 » par « ligne ». Je réalise le même traitement que tout à l'heure, à savoir mettre la valeur « vide » dans chacune des cases. Je corrige un peu mon indentation. Donc j'obtiens au final un ensemble de lignes de codes qui vont me permettre d'itérer sur l'ensemble des lignes pour en modifier le contenu. Vous noterez au passage également l'utilisation du passage par référence pour la variable « d'itération sur ensemble de valeurs » afin de garantir que la grille va effectivement être modifiée. [inaudible]

notes

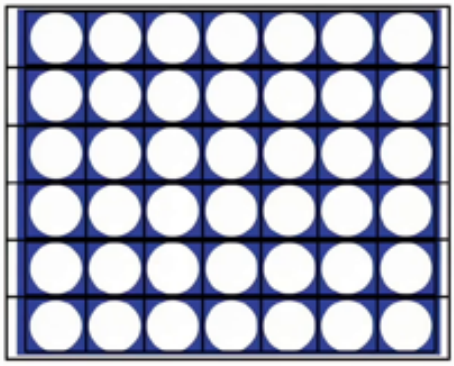
résumé

7m 25s

Fonction affiche

```
void affiche(

Grille grille;
initialise(grille);
...
affiche(grille);
```



Nous savons désormais initialiser une grille qui a été au préalable déclarée au moyen du type « grille » qui a été défini à l'étape précédente. Intéressons-nous maintenant à l'affichage de la grille. Comme tout à l'heure, imaginons l'utilisation concrète de cette fonction d'affichage. Il est toujours bon de commencer à imaginer comment la fonction va être utilisée avant de s'intéresser à son codage effectif. Typiquement on peut imaginer qu'une fois la grille initialisée, réalisés d'autres traitements sur la grille, je puisse avoir besoin de l'afficher. Il est naturel d'imaginer que simplement j'invoque une fonction « affiche » qui prend en paramètre la grille « affiche ». Intéressons-nous maintenant au prototype. Comme tout à l'heure, un type « void » est naturel car une fonction d'affichage va simplement afficher la grille sur le terminal et n'a pas besoin de me retourner de résultat. Donc ici je lui passe la grille. Maintenant va se poser la question de : « Sous quelle forme passer le paramètre « grille » ? » Est-ce que je fais un passage par valeur ou par référence ? La fonction d'affichage a pour vocation d'afficher les valeurs des différentes cases de la grille de jeux. Elle n'a donc certainement pas pour but de modifier le contenu de cette grille.

notes

résumé

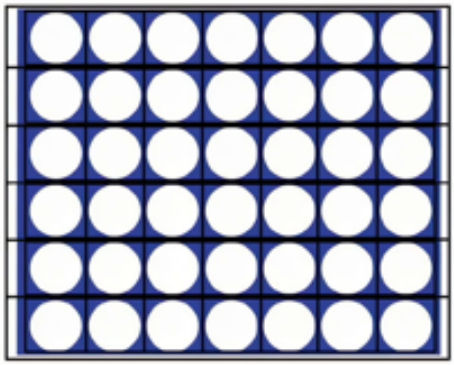
8m 13s

Fonction affiche

```
void affiche(

Grille grille;
initialise(grille);
...
affiche(grille);
```

void affiche (Gri



Du coup, il est tout à fait naturel d'imaginer que le passage du paramètre « grille » va se faire par valeur.

notes

résumé

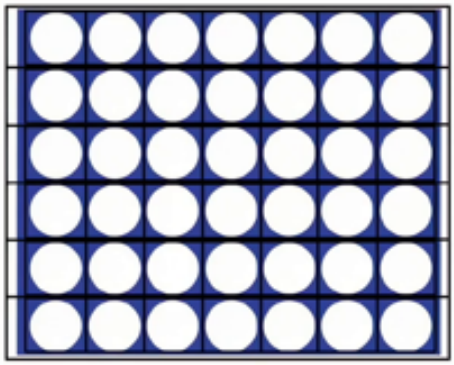
9m 23s

Fonction affiche

```
void affiche(

Grille grille;
initialise(grille);
...
affiche(grille);
```

void affiche (Grille grille);



Parce qu'il n'est pas nécessaire, la fonction « affiche » modifie le contenu de la grille. Nous avons vu cependant que lors d'un passage par valeur, ce qui se produit

notes

résumé

9m 29s

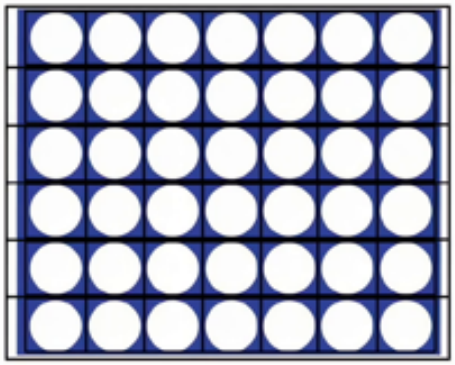
Fonction affiche

```
void affiche(

Grille grille;
initialise(grille);
...
affiche(grille);
```

void affiche (Grille grille) ;
↳ copie

void affiche (Grille & grille)



c'est qu'il y a une copie de l'argument passé à la fonction dans une zone locale à la fonction « affiche ». Si l'objet passé en argument est un objet volumineux, cette copie peut être relativement coûteuse. A ce moment là, on a recours à une alternative qui est une optimisation qui consiste à passer le paramètre par référence.

notes

résumé

9m 38s

Fonction affiche

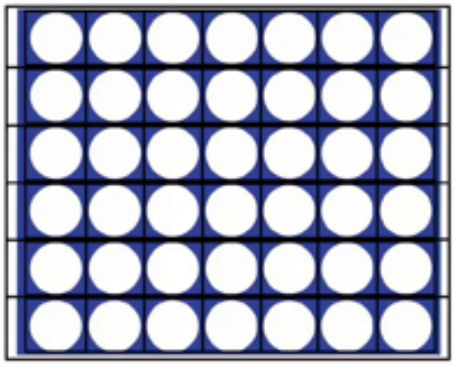
```
void affiche(

Grille grille;
initialise(grille);
...
affiche(grille);
```

Void affiche (Grille grille) ;
↳ copie

Void affiche (const Grille & grille)

Void



Ce qui va autoriser la fonction « affiche » à travailler directement sur la zone mémoire qui correspond à l'argument qui lui est passé. Sans faire de copie. Mais on prend la précaution de le passer par référence constante afin de garantir que la fonction « affiche », même si elle a le droit d'opérer directement sur l'argument, ne va pas pour autant en modifier le contenu. Si par exemple dans le corps de la fonction « affiche » je commence à modifier le contenu, et bien j'aurais un message d'erreur à la compilation.

notes

résumé

10m 1s

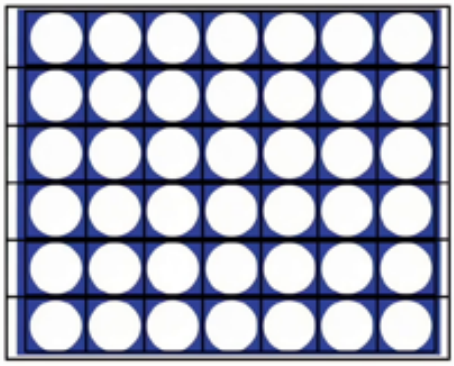
Fonction affiche

```
void affiche(

Grille grille;
initialise(grille);
...
affiche(grille);
```

void affiche (Grille grille) ;
↳ copie

void affiche (const Grille & grille)



Notez qu'on aurait parfaitement pu écrire ceci : Il s'agit d'une notation parfaitement équivalente à la précédente. On peut utiliser indifféremment l'une ou l'autre des notations.

notes

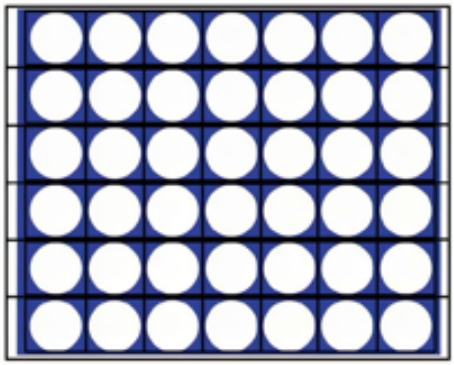
résumé

10m 28s

Fonction affiche

```
void affiche(const

Grille grille;
initialise(grille);
...
affiche(grille);
```



Il s'agit ici d'une optimisation qui va permettre à la fonction de travailler directement sur l'argument mais en mettant en place un certain nombre de garde-fou qui empêchent quand même la fonction de modifier le contenu puisque ce n'est pas sa vocation. Cette précaution était indispensable en C++98. Elle ne l'est pas systématiquement dans tous les cas en C++2011 Cependant, les considérations appliquées vont au-delà du cadre de ce cours. Nous allons simplement ici nous contenter de la mettre en œuvre comme précaution et comme bonne pratique pour éviter la copie.

notes

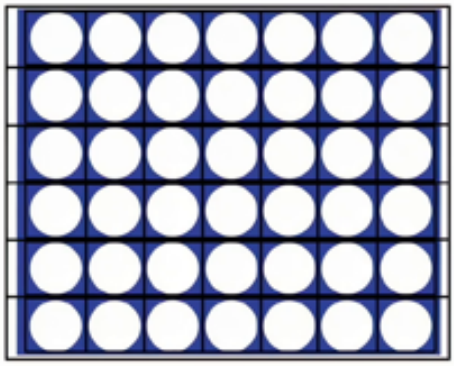
résumé

10m 38s

Fonction affiche

```
void affiche(const Grille& grille)
{
  for(auto ligne : grille) {
    f

Grille grille;
initialise(grille);
...
affiche(grille);
```



Ici, le prototype de notre fonction « affiche » va utiliser le passage par référence constante pour éviter la copie. Procédons maintenant au codage du corps de la fonction « affiche ». Ici le codage est assez naturel. Il ressemble beaucoup à ce que nous avons fait pour la fonction « initialise ». Il s'agit d'itérer sur chacune des lignes de la grille et pour chacune des lignes de la grille, itérer sur chacune des cases pour en afficher le contenu sous un format particulier. Ici nous n'allons pas procéder à l'affichage graphique nous allons simplement afficher le contenu de chaque cellule selon une convention particulière sur le terminal. Pour itérer sur l'ensemble des lignes de la grille, une « itération sur ensemble de valeurs » est tout à fait naturel. Je sais exactement combien de lignes, combien de colonnes a ma grille. Donc ici, pour chacune des lignes de la grille, ce qui s'écrit comme ceci avec une « itération sur ensemble de valeurs ».

notes

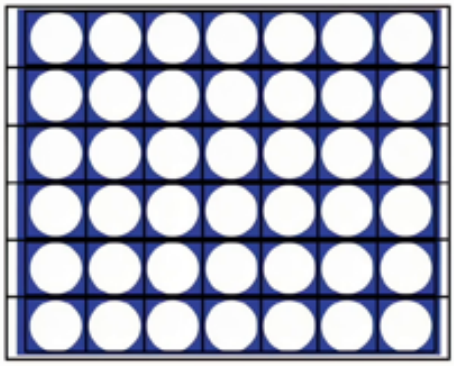
résumé

11m 11s

Fonction affiche

```
void affiche(const Grille& grille)
{
  for(auto ligne : grille) {
    for(a
```

```
Grille grille;
initialise(grille);
...
affiche(grille);
```



Je vais itérer sur chacune des cases de la ligne en question.

notes

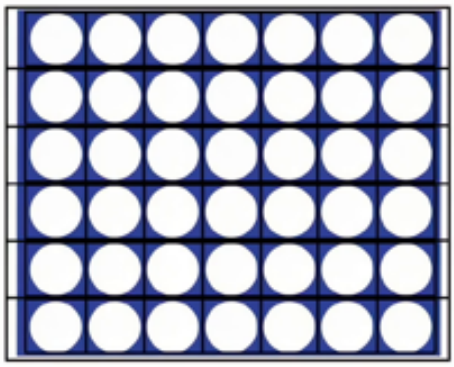
résumé

12m 1s

Fonction affiche

```
void affiche(const Grille& grille)
{
  for(auto ligne : grille) {
    for(auto kase : ligne) {
      if (kase == vide) {
        cout << ' ';
      } else if (kase == rouge) {
        cout << |

Grille grille;
initialise(grille);
...
affiche(grille);
```



Donc pour chacune de ces cases, en afficher le contenu sous un format particulier. Ici je dois procéder à des tests pour savoir, en fonction du contenu, ce qu'il faut afficher exactement. Par exemple, si la case contient la valeur « vide », donc la valeur du type énuméré « Couleur » est « vide », alors je peux choisir, par convention, d'afficher sur le terminal un espace, sinon procéder à d'autres tests. Si la valeur de la case correspond à la valeur du type énuméré « rouge », seconde convention possible -- je peux choisir une convention quelconque --

notes

résumé

12m 2s

ici par exemple 'O' pour la valeur « rouge ». Et puis dernier cas, je sais qu'il s'agit de la valeur « jaune »; je fais un choix d'affichage quelconque, par exemple 'X' pour afficher la valeur « jaune ». Ici, se termine le corps de ma première « boucle for », la « boucle for » la plus interne. A ce stade j'ai affiché le contenu d'une ligne. Je prends la précaution de faire un saut de ligne pour séparer les différentes lignes une fois qu'une ligne a été entièrement affichée. Ce qui clôt le corps de la toute première « boucle for ». Ensuite je peux terminer le corps de ma fonction « affiche ». Vous noterez ici qu'il n'a pas été nécessaire dans mes « itérations sur ensemble de valeurs » d'avoir recours au symbole &. Pourquoi ? Parce que je ne fais que consulter le contenu des différents éléments. Je ne veux pas en modifier le contenu Dans ce cas là, « auto » sans le symbole du passage par référence est entièrement suffisant. En bon programmeur, je vais maintenant m'atteler à commenter ma fonction « affiche ». En effet, les choix d'affichage auxquels j'ai procédé ne sont pas forcément prévisibles, donc il est bien que quelqu'un qui consulte mon code puisse connaître ces conventions sans pour autant aller lire l'intégralité du corps de la fonction. A ce moment-là, je prends la précaution de commenter mon code en précisant les conventions adoptées pour le codage. Je précise que je vais afficher un O pour une case rouge et X pour une jaune.

notes

résumé

12m 37s

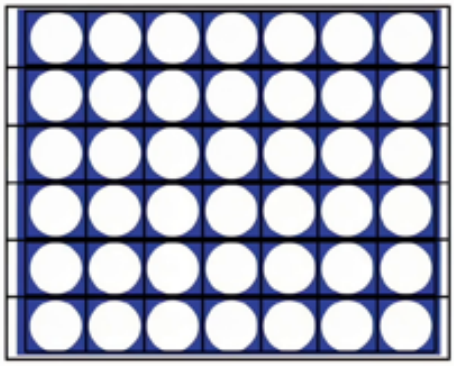
Tester initialise et affiche

EPFL

```
int main()
{
  Grille grille;

  initialise(grille);
  gri
  affiche(grille);

  return 0;
}
```



Tester les fonctionnalités au fur et à mesure que l'on progresse dans le codage d'une application est absolument indispensable pour produire du code robuste et de qualité. C'est une recommandation qui vous a été faite tout au début de notre étude de cas sur le Puissance 4. Nous allons donc maintenant commencer à tester nos premières fonctionnalités, à savoir l'initialisation de la grille et son affichage. Pour tester les fonctionnalités en question, il me suffit d'écrire un petit programme principal. qui va simplement invoquer les fonctions « initialise » et « affiche » sur une grille que j'aurais au préalable déclarée. Ici, j'invoque ma fonctionnalité d'initialisation sur la grille et l'affichage de la grille. Il peut être naturel ici d'avoir envie de tester les affichages sous toutes les formes possibles. Si je me contente d'initialiser la grille puis de l'afficher directement je n'aurais que des cases vides.

notes

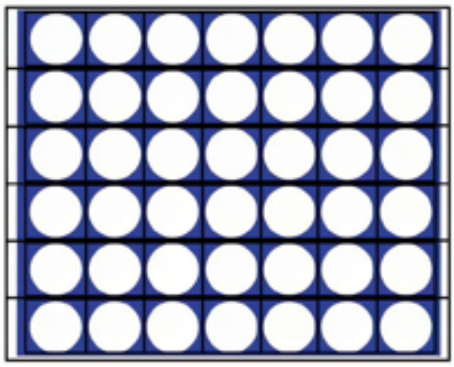
résumé

14m 6s

Retour sur affiche

```
// affiche 0 pour une case rouge, X pour une case jaune
void affiche(const Grille& grille)
{
  for(auto ligne : grille) {
    for(auto kase : ligne) {
      if (kase == vide) {
        cout << ' ';
      } else if (kase == rouge) {
        cout << '0';
      } else {
        cout << 'X';
      }
    }
}
```

```
␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣
␣␣␣X0␣␣␣
␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣
```



A ce moment-là, il est intéressant de voir que tous les cas sont correctement traités Je vais essayer de couvrir tous les cas en remplissant ma grille à certains endroits avec des jetons jaunes et à d'autres avec des jetons rouges. Ce que je fais ici par exemple sur des cases arbitrairement choisies. L'affichage auquel je m'attends, suite à l'exécution de ce programme, devrait ressembler à ceci. Ici pour toutes les cases de la grille, hormis celle que j'aurais explicitement affectées, j'aurais l'affichage d'un espace. Ce que je représente ici pour plus de clarté par ce symbole, mais il faut noter qu'en réalité, rien ne sera affiché, il n'y aura qu'un espace. Et puis, pour les deux cases particulières que j'ai remplies avec les jetons jaunes et rouges, j'aurai des affichages particuliers. Ici la case de la grille à la ligne 2 et à la colonne 3 doit être jaune, ce qui correspond à ma convention d'affichage X. Ici j'ai bel et bien l'indice de ligne zéro, un, deux... Ce qui correspond à ce numéro de ligne. L'indice de colonne 0, 1, 2, 3 ce qui correspond à ceci. De façon analogue je dois donc avoir la convention d'affichage pour un jeton rouge à la position ligne 2 colonne 4. Concrètement dans mon jeu du puissance 4, ce jeu de test me permet de créer une grille qui aurait un jeton jaune ici et un jeton rouge là.

notes

résumé

15m 1s

Retour sur affiche

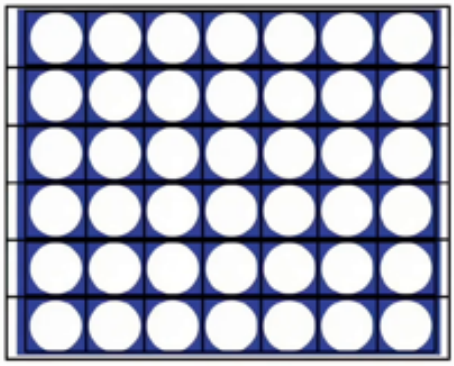
```
// affiche 0 pour une case rouge, X pour une case jaune
void affiche(const Grille& grille)
{
  cout << endl;

  for(auto ligne : grille) {
    cout << " |";
    for(auto kase : ligne) {
      if (kase == vide) {
        cout << ' ';
      } else if (kase == rouge) {
        cout << '0';
      } else {
        cout << 'X';
      }
      cout << '|';
    }
    cout << endl;
  }

  cout << '=';
  for(size_t i(1); i <= grille[0].size(); ++i) {
    cout << '=' << i;
  }
  cout << "==";

  cout << endl << endl;
}
```

```
 | | | | | | | |
 | | | | | | | |
 | | | |X|0| | |
 | | | | | | | |
 | | | | | | | |
 | | | | | | | |
==1=2=3=4=5=6=7==
```



L'essentiel est maintenant mis en place et on peut commencer à s'intéresser à des aspects un petit peu plus cosmétiques, des aspects de peaufinage. Ici l'affichage de notre grille est relativement austère. On ne peut pas bien distinguer les différentes colonnes entre elles. Et ce d'autant qu'on ne voit pas très bien la séparation entre les espaces. Il serait bien d'améliorer un petit peu cet affichage ce que l'on peut faire en complétant le code existant, en y greffant les ajouts suivants que je vais maintenant commenter. Ceci nous permettrait d'avoir un affichage un petit peu plus explicite. Notamment de séparer les différentes colonnes par des barres verticales et puis, de numéroter explicitement les différentes colonnes en-dessous de la grille afin de pouvoir visualiser plus clairement le contenu de notre grille de jeu. Quels ont été les ajouts nécessaires pour parvenir à cette forme d'affichage ? Et bien, je commence avant chaque ligne, par afficher un espace suivi d'une barre verticale. Ce qui va permettre d'afficher chacun de ces éléments pour chacune des lignes. Ensuite, après l'affichage du contenu de chaque case

notes

résumé

16m 32s

qu'il soit vide, rouge ou jaune, je suis l'affichage par celui d'une barre verticale. Ce qui me permet d'afficher chacune de ces barres verticales ici. Le reste des traitements est exactement pareil. Ensuite lorsque j'ai affiché l'intégralité de la grille, je vais m'intéresser à afficher cette partie en dessous de la grille, ce qui fait l'objet des lignes de codes que vous avez ici sous les yeux et que je vais commenter. Je commence par faire afficher un symbole égal qui est ce symbole, qui est un cas particulier en fait, qui ne correspond pas au contenu d'une colonne. Il correspond simplement à l'espace précédant la première colonne. Ensuite par une « boucle for », je vais itérer sur l'ensemble des colonnes pour afficher le numéro. Je précède l'affichage de chaque numéro par l'affichage du symbole égal. En fait, la « boucle for » ici va me permettre d'afficher chaque numéro précédé du signe égal. Donc ici, j'ai pris la précaution de numéroter mes colonnes à partir du nombre 1 afin que ce soit plus lisible. Une fois que j'ai terminé cette « boucle for », j'aurai affiché le dernier numéro de colonne précédé du symbole égal. Si je veux clore mon affichage de façon harmonieuse, il ne me reste plus qu'à afficher le double symbole égal que vous avez ici sous les yeux. Ceci est suivi d'autant de retours de ligne qu'il me semble nécessaire. A ce moment-là, j'aurais peaufiné mon affichage de sorte à le rendre plus lisible. Notez ici que je ne suis pas en train d'itérer sur l'ensemble des valeurs de la grille. Mais je suis en train d'itérer autant de fois qu'il y a de colonnes dans la grille. A ce moment-là, j'utilise une itération for classique et non pas une « itération

notes

résumé

17m 37s

sur ensemble de valeurs ». Mon « i » prend autant de valeurs qu'il y a d'éléments, de nombre de colonnes. Le nombre de colonnes est un élément de type « size », ce que je représente en C++ au moyen du type « size_t ». Il s'agit ici d'une bonne pratique à observer pour coder ce genre de traitement. Nous voici arrivés au terme de cette étape. Nous savons désormais modéliser une grille de jeu Puissance 4. Nous savons également l'initialiser et en afficher le contenu. Nous pouvons maintenant passer au vif du sujet qui va être l'objet des séquences suivantes. des séquences suivantes.

notes

résumé