Support de cours

Cours:

Initiation à la programmation (en C++)

Vidéo:

Puissance 4 - fonction joue variantes et révision (partie 1)

Concepts (extraits des sous-titres générés automatiquement) :

Numéro de colonne. Variable pleine. Sortir du tableau. Version alternative. Vidéo précédente. Nombre de lignes de commentaires. Problème de départ. Version précédente. Dernier niveau. Première version. Code de façon. Code de cette première version. Couleur du joueur. Réflexe de façon systématique. Différentes versions alternatives de la fonction.

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

vers la vidéo

EPFL

Puissance 4 : fonction `joue` variantes et révision

(Partie 1)

Initiation à la programmation (C++)

Vincent Lepetit, Jean-Cédric Chappelier et Jamila Sam

...

notes

résumé

0m 0s

Dans cette vidéo nous allons présenter les différentes versions alternatives

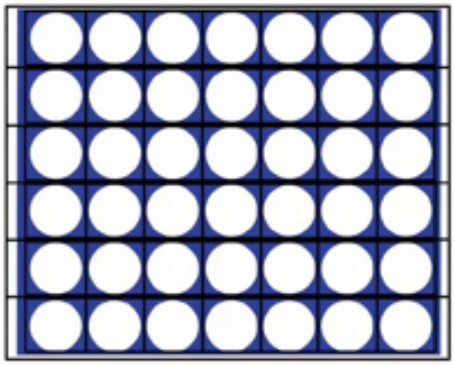
notes

résumé

0m 1s

Une version alternative de la fonction `joue`

```
bool joue(Grille& grille, size_t colonne, Couleur couleur)
{
  // si |
}
```



de la fonction « joue » que je vous ai présenté dans la vidéo précédente. Je vous rappelle que cette fonction « joue » a pour but de placer un pion dans la grille, étant donné donc la grille dans laquelle on joue le numéro de colonne dans lequel on veut insérer le pion dans lequel on veut jouer le pion et la couleur du joueur, je vous rappelle aussi que cette fonction vérifie la validité du coup c'est-à-dire que si la colonne est pleine on va dire que le coup n'est pas valide. Pour avoir une version alternative le prototype tout d'abord, quelle que soit l'implémentation de la fonction « joue », le prototype ne va donc pas changer, on aura toujours ici une grille par référence on aura toujours une colonne à passer et une couleur donc rien de changé ici mais l'idée est d'avoir un algorithme assez différent

notes

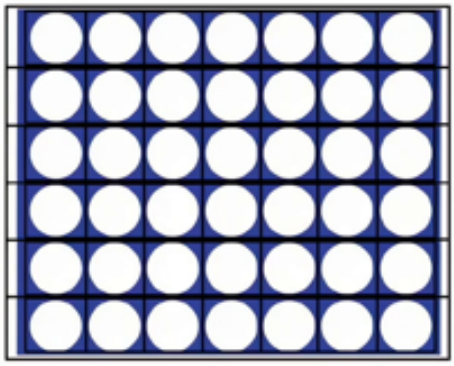
résumé

0m 5s

Une version alternative de la fonction `joue`

```
bool joue(Grille& grille, size_t colonne, Couleur couleur)
{
  // si la colonne est pleine, le coup n'est pas valide :
  if (grille[0][colonne] != vide) {
    return false;
  }

  // on parcourt la colonne en partant du bas jusqu'à trouv
}
```



et de commencer par se dire que si la colonne est pleine alors déjà on ne joue pas. Donc l'idée est vraiment de commencer par là plutôt que de le tester après donc ça s'écrit que si, comme on va tester que la colonne est pleine, je rappelle que une colonne est pleine si le dernier niveau le niveau 0 est occupé donc ici on va tester que si la grille est en position 0 ici, c'est bien cette ligne là et dans la colonne qu'on a reçu n'est pas vide. A ce moment-là ça veut dire que la colonne est pleine et donc de suite on retourne le fait que le coup est invalide c'est-à-dire on retourne « false ». Nous voilà donc dégagés de cette condition et donc on peut maintenant tranquillement chercher l'endroit, puisqu'on sait que la colonne n'est pas pleine, on peut donc maintenant tranquillement chercher l'endroit où on va jouer pour ça on parcourt la colonne comme dans la version précédente

notes

résumé

0m 50s

en partant du bas donc cela s'écrit en C++ en déclarant donc une variable qui va nous permettre de parcourir les lignes comme dans la version précédente de la fonction « joue » et qu'on l'initialise à la taille de la grille -1 puis ensuite la position courante. C'est-à-dire que la « grille[ligne][colonne] » n'est pas vide alors on décrémente la ligne de 1. Enfin pour terminer on remplit la case qu'on a trouvé puisqu'on sait qu'on peut en trouver une et enfin on renvoie « true » puisqu'on a pu jouer le coup.

notes

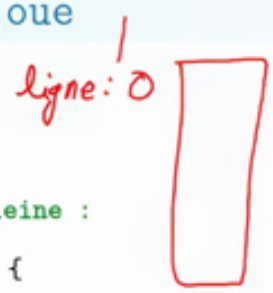
résumé

1m 49s

Encore une autre version de la fonction joue

```
bool joue(Grille& grille, size_t colonne, Couleur couleur)
{
  // on parcourt la colonne en partant du bas...
  size_t ligne(grille.size() - 1);
  // ...jusqu'à trouver une case vide...
  // ...ou jusqu'en haut de la colonne si la colonne est pleine :
  bool pleine(false);
  while ((not pleine) and (grille[ligne][colonne] != vide)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  // si on n'est pas arrivé jusqu'en haut de la colonne, on remplit la case vide trouvée,
  // sinon c'est que la colonne est pleine et le coup n'est pas valide :
  if (not pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
}
```



Voilà il faudrait à ce stade bien sûr tester cette nouvelle fonction « joue » ce que je vous laisse faire en tant qu'exercice, je vous laisse aussi apprécier la différence entre les deux codes indépendamment d'ailleurs du nombre de lignes de commentaires. Voyons maintenant une troisième version de cette fonction « joue » qui repart justement de la première version que nous avons présenté donc dans la vidéo précédente et qu'on va essayer d'améliorer, donc je vous rappelle ici le code de cette première version, et l'idée est de se concentrer sur ce qui nous a posé initialement souci et aporté donc tant de lignes avec là « pleine », révisé aussi ici, qui était donc justement sur ce point de décrémenter ici « ligne » quand « ligne » valait 0. Qu'est ce qui se passe à ce moment là ? Est-ce qu'on pourrait peut-être écrire le code de façon un petit peu différente ? Donc l'idée c'est de supprimer ce booléen « pleine » et de le remplacer par un autre test. Nous reprenons depuis le problème de départ, peut-être sur une question que nous avions laissé en suspens qui est que, je vous rappelle que ce qui se passait c'est que quand la ligne était nulle on faisait quand même, puisqu'on avait à ce moment-là pas mis tout ce texte ici, on faisait quand même un « ligne-- »

notes

résumé

2m 28s

et donc qu'est-ce qui se passe dans ce cas-là, on risquait de sortir du tableau. Donc pour ça on va revenir sur le type de ligne qui est un type « size_t » et dans le cours qui vous présentait ce type on vous a dit que un « size_t »c'est toujours un entier positif. Donc on a un problème ici de représentation de nombre.

notes

résumé

3m 49s

Encore une autre version de la fonction joue

```
bool joue(Grille& grille, size_t colonne, Couleur couleur)
{
  // on parcourt la colonne en partant du bas...
  size_t ligne(grille.size() - 1);
  // ...jusqu'à trouver une case vide...
  // ...ou jusqu'en haut de la colonne si la colonne est pleine.
  // Si le test (ligne < grille.size()) devient faux, c'est qu'on a
  // soustrait 1 à ligne quand elle valait 0, ce qui arrive quand la
  // colonne est pleine.
  while ((ligne < grille.size()) and (grille[ligne][colonne] != vide)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  // si on n'est pas arrivé jusqu'en haut de la colonne, on remplit la case vide trouvée,
  // sinon c'est que la colonne est pleine et le coup n'est pas valide :
  if (not pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
```

Et donc ce qui va se passer en pratique en mémoire c'est qu'on va avoir un très grand nombre. Peu importe les aspects techniques ici ; c'est pas ça qui nous intéresse ! Mais ce très grand nombre ne va certainement pas être un numéro de ligne de notre tableau. Et donc ce qu'on devrait faire -- ce qui devrait d'ailleurs être le réflexe de façon systématique ; et qui va me donner une autre critique sur ce code qu'on a écrit ici plus tard -- ce qu'on devrait faire, donc, c'est de tester systématiquement que les accès à un tableau sont corrects ; en tout cas de le garantir, soit par une démonstration sur l'algorithme, soit par un vrai test. Et donc quelles sont ces conditions ici ? C'est qu'on doit absolument avoir que « ligne » doit être inférieur à « grille.size ». Et, de même, un index de tableau ne peut pas être plus grand que la taille de ce tableau. Et, de même, on devrait avoir et vérifier -- ce qu'on discutera un peu plus tard -- que « colonne » doit être inférieur à « grille[ligne().size] ». Donc que cette valeur ici « -1 », c'est-à-dire « size_t 0 » sur lequel on fait « -- » , soit représentée d'une façon ou d'une autre, par exemple « impossible », ou par exemple un très grand nombre, toujours est-il que ce nombre qui est « 0-- » c'est-à-dire -1 n'est pas représentable sur un « size_t » et donc la condition de grille ici inférieure à grille.size ne sera pas vérifiée. C'est donc cette condition que nous allons utiliser comme condition de débordement de notre tableau et donc on va l'écrire ici avec un commentaire donc si le test de « ligne inférieur à grille.size » est faux, alors c'est que la colonne était pleine et

notes

résumé

4m 8s

Encore une autre version de la fonction joue

```
bool joue(Grille& grille, size_t colonne, Couleur couleur)
{
  // on parcourt la colonne en partant du bas...
  size_t ligne(grille.size() - 1);
  // ...jusqu'à trouver une case vide...
  // ...ou jusqu'en haut de la colonne si la colonne est pleine.
  // Si le test (ligne < grille.size()) devient faux, c'est qu'on a
  // soustrait 1 à ligne quand elle valait 0, ce qui arrive quand la
  // colonne est pleine.
  while ((ligne < grille.size()) and (grille[ligne][colonne] != vide)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  // si on n'est pas arrivé jusqu'en haut de la colonne, on remplit la case vide trouvée,
  // sinon c'est que la colonne est pleine et le coup n'est pas valide :
  if (not pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
```

qu'il faut donc s'arrêter, que le coup est invalide. Maintenant qu'on a bien compris et expliqué ce que l'on voulait faire on peut maintenant corriger notre code donc on supprime la variable pleine, on supprime ici le test qui l'impliquait et on le remplace par le test que la ligne donc est inférieure à la taille de la grille. Notez que l'on a donc bien ici cette protection de l'indice d'accès par rapport à la taille du tableau accédé avant d'y accéder. Ce qui est un conseil à vérifier de façon systématique. Si c'est garanti par une boucle « for » ou autre, vous n'allez bien sûr pas remettre le test à chaque fois, mais vous devriez avoir comme réflexe chaque fois que vous faites un accès comme ça à un tableau au travers d'un indice de vous demander : est-ce que la condition qu'il est strictement plus petit que la taille de ce tableau est vérifiée ? Donc là on voit bien qu'on l'a explicitement écrit et qui nous sert de test très utile

notes

résumé

pour tester si la colonne était pleine. Ensuite on peut donc continuer à supprimer tout ce qui dépend de « pleine » et enfin corriger la fin de notre programme ; où on n'a plus besoin non plus de cette variable « pleine ». Donc ici toujours le même test, si la ligne est plus petite que la taille de la grille alors c'est qu'on peut jouer sinon c'est qu'on n'a pas pu jouer. Et, on modifie donc le code en conséquence. Ce qui termine notre troisième version de la fonction « joue ». Ceci dit, si vous avez bien suivi ce que je viens de vous dire dans cette dernière fonction il reste une erreur à toutes nos fonctions « joue ». Est-ce que vous voyez ce qu'il faudrait encore ajouter pour que ces trois fonctions « joue » soient correctes. soient correctes.

notes

résumé

6m 49s