Support de cours

Cours:

Initiation à la programmation (en C++)

Vidéo:

Puissance 4 - finalisation

Concepts (extraits des sous-titres générés automatiquement) :

Existence d'une fonction. Programme principal. Situations de match nul. Couleur du joueur alterne. Première chose naturelle. Terme de notre étude de cas. Première case disponible. Fin de chaque itération de la boucle. Couleur du joueur courant. Codage de la fonction. Types essentiels. Objet de notre séquence. Pions jaunes. Dernier point essentiel. Codage de notre petite application.

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

vers la vidéo

EPFL

Puissance 4 : finalisation

Initiation à la programmation (C++)

Vincent Lepetit, Jean-Cédric Chappelier et Jamila Sam

...

notes

résumé

0m 0s

Nous sommes arrivés à ce stade presqu'au terme

notes

résumé

0m 1s

Retour sur la fonction `main`

```
    // on change la couleur pour la couleur de l'autre joueur:
    if (couleur_joueur == jaune) {
      couleur_joueur = rouge;
    } else {
      couleur_joueur = jaune;
    }
  } while(not gagne and not plein(grille));

  if (gagne) {
    // attention, on a change la couleur pour la couleur de l'autre joueur !
    if (couleur_joueur == jaune) {
      cout << "Le joueur O a gagne !" << endl;
    } else {
      cout << "Le joueur X a gagne !" << endl;
    }
  } else {
    cout << "

  return 0;
}
```

de notre étude de cas du « Puissance 4 ». Nous savons faire jouer en alternance les deux joueurs, tester si un joueur a gagné. Il nous reste maintenant à peaufiner un petit peu le programme principal, pour indiquer notamment aux deux joueurs, qui a gagné et pour faire face aux situations de match nul. C'est l'objet de notre séquence d'aujourd'hui. Pour rappel, notre jeu de « Puissance 4 » tel qu'on le connait jusqu'ici, c'est faire jouer en boucle deux joueurs, le joueur 'X' qui joue les pions jaunes, et le joueur 'O' qui joue les pions rouges. Ces deux joueurs jouent en alternance. C'est-à-dire qu'à la fin de chaque itération de la boucle, nous garantissons que la couleur du joueur alterne entre le jaune et le rouge. Et les deux joueurs jouent jusqu'à ce que l'un des deux ait gagné. Il nous reste cependant à finir un petit peu ce programme. La première chose naturelle à faire est, lorsque la boucle a terminé son exécution, ce qui signifie que l'un ou l'autre des joueurs à gagné, de signaler qui a effectivement gagné la partie de « Puissance 4 ». Une seconde chose à faire est de considérer la situation où aucun des deux joueurs n'a gagné, ce qui n'est pas couvert dans le cas actuel, et qui ne correspond pas à une situation très réaliste. Nous allons nous occuper de ceci maintenant. Commençons par faire en sorte que notre programme indique quel est le joueur gagnant, donc après la boucle, nous devons ajouter un certain nombre d'instructions, qui consistent naturellement à tester la couleur du joueur courant, et si cette couleur est jaune par exemple, afficher que le joueur gagnant est le joueur rouge. Alors pourquoi le joueur rouge ici et non pas le joueur jaune ? Eh bien pour rappel, à la fin de l'itération de la boucle, nous alternons d'un joueur à l'autre, et donc si

notes

résumé

0m 5s

Retour sur la fonction `main`

```
    // on change la couleur pour la couleur de l'autre joueur:
    if (couleur_joueur == jaune) {
      couleur_joueur = rouge;
    } else {
      couleur_joueur = jaune;
    }
  } while(not gagne and not plein(grille));

  if (gagne) {
    // attention, on a change la couleur pour la couleur de l'autre joueur !
    if (couleur_joueur == jaune) {
      cout << "Le joueur O a gagne !" << endl;
    } else {
      cout << "Le joueur X a gagne !" << endl;
    }
  } else {
    cout << "

  return 0;
}
```

la couleur du joueur courant est la couleur jaune, ceci signifie que le joueur qui a remporté la partie, était celui de la couleur alternative, donc le rouge ; et donc ça signifie que c'est effectivement bien le rouge qui a gagné dans le cas où on a alterné à la couleur jaune. Ce qui explique la raison pour laquelle nous indiquons ici que si la couleur du joueur courant est jaune, c'est le joueur rouge qui a effectivement gagné. Et pareil, nous couvrons la situation où c'est le joueur jaune qui a gagné. Et nous avons ainsi couvert les deux cas possibles. Nous prenons la précaution de commenter ce test un petit peu particulier, donc en indiquant qu'effectivement on a au préalable changé la couleur du joueur courant, ce qui explique la nature un petit peu contre-intuitive de ce test. Alors comme indiqué en introduction de cette séquence, il existe aussi des situations où les deux joueurs vont jouer l'un après l'autre et vont remplir totalement la grille sans que l'un ou l'autre n'ait gagné ; ce qui va correspondre à une situation de match nul. Il nous faut donc parer à cette situation également ; et donc sortir de la boucle, si l'un des deux a gagné ou si la grille est pleine. Donc nous devons compléter notre code, de sorte à ce que ce cas soit également couvert. Si nous ne couvrons pas ce cas, le programme aura un comportement erroné, qui fera que, une fois la grille remplie, le programme va tourner en boucle, en demandant sans cesse au même joueur d'indiquer un numéro de colonne à jouer alors que la grille est complètement remplie et qu'il n'est plus possible de placer aucun jeton. Ceci correspond bien sûr à une situation erronée à laquelle il faut remédier. Donc les modifications à apporter au programme pour parer à cette situation, sont assez naturelles. Donc nous devons indiquer

notes

résumé

Retour sur la fonction `main`

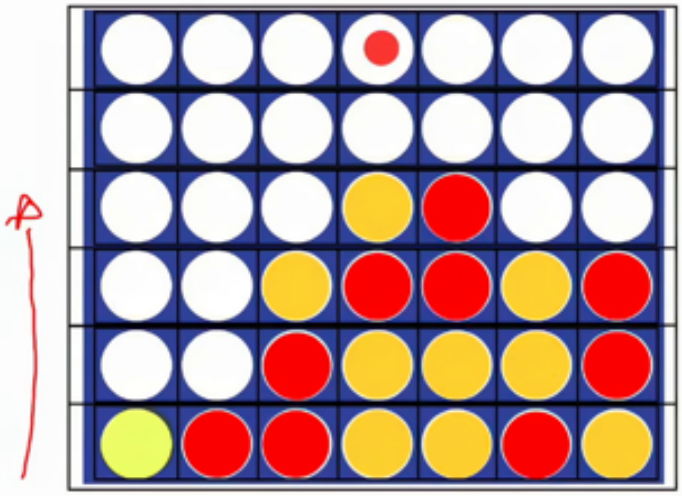
```
    // on change la couleur pour la couleur de l'autre joueur:
    if (couleur_joueur == jaune) {
      couleur_joueur = rouge;
    } else {
      couleur_joueur = jaune;
    }
  } while(not gagne and not plein(grille));

  if (gagne) {
    // attention, on a change la couleur pour la couleur de l'autre joueur !
    if (couleur_joueur == jaune) {
      cout << "Le joueur O a gagne !" << endl;
    } else {
      cout << "Le joueur X a gagne !" << endl;
    }
  } else {
    cout << "

  return 0;
}
```

que nous itérons, que nous continuons à boucler dans le cas où aucun des deux joueurs n'a gagné, ce qui est déjà testé, et tant que la grille n'est pas pleine, ce qui peut s'écrire naturellement de cette façon. Donc nous anticipons l'existence d'une fonction « plein » qui va tester si la grille est pleine ou non. Ceci présuppose que l'on apporte aussi quelques modifications après l'exécution de la boucle. En effet à ce stade-là, nous n'aurons plus à la sortie de la boucle une situation unique où fatalement l'un ou l'autre des joueurs a gagné, mais il peut y avoir également une situation de match nul qu'il faudra signaler. Donc il faut prévoir de modifier le programme de sorte à ce que, si il y a effectivement un gagnant, alors effectivement ici, nous sommes dans la situation où il faut signaler quel est le gagnant, mais ça n'est pas fatalement le cas, il peut y avoir la situation où il n'y a pas de gagnant,

notes

résumé

Fonction plein



et à ce moment là, il faut indiquer qu'il y a match nul. Et nous avons ainsi couvert l'ensemble des situations possibles. Passons maintenant au codage de la fonction « plein ». Donc pour rappel, le programme va demander en alternance au joueur jaune et au joueur rouge, sur quelle colonne il veut placer son jeton. Supposons que la main soit au joueur jaune, si par exemple il décide d'introduire un jeton à la première colonne, eh bien ce remplissage va se faire toujours de bas en haut, c'est-à-dire que nous allons mettre automatiquement le jeton sur la première case disponible, en partant du bas. Ceci a pour conséquence que pour tester si la grille est pleine, il n'est pas nécessaire de tester l'intégralité des cases de la grille ; pour savoir si elles sont occupées par une couleur particulière, il suffit simplement de tester les cases de la toute première ligne. En effet, dans le cas où la grille est pleine, on aura rempli l'intégralité des colonnes en partant du bas. Ce qui signifie que l'ensemble des cases de la première ligne,

notes

résumé

4m 1s

Fonction plein

```
// plein(grille)
bool plein(const Grille& grille)
```

auront également été remplies. Passons maintenant au codage effectif de la fonction « plein ». Donc au niveau méthodologique, nous nous étions demandé de façon rapide, parce qu'il s'agit d'une fonction très simple, ce qui concerne le « quoi » : que fait cette fonction « plein » ? Dans le programme principal, nous avions prévu que cette fonction, prendrait en argument la grille et testerait si elle est pleine ou non. Ce qui correspond assez naturellement au prototype suivant : donc la fonction retourne un « boulean » indiquant que la grille est pleine ou non, et prend en paramètre, une grille. Dans les séquences précédentes, nous avons vu que pour toutes les fonctions qui avaient en paramètre, un objet relativement volumineux, qui n'était utilisé qu'en lecture, qu'en consultation, qu'on ne veut pas modifier, il était judicieux de faire un passage par référence constante. Nous reprenons exactement ici cette précaution. Ici la fonction « plein » a pour vocation de consulter le contenu de certaines cases de la grille, pour savoir si elles sont remplies ou non,

notes

résumé

5m 1s

Fonction plein

```
// plein(grille)
bool plein(const Grille& grille)
{
  // Si on trouve une ca
```

et n'a pas pour vocation de modifier la grille. Et c'est la raison pour laquelle nous faisons également ici, un passage par référence constante. Passons maintenant au « comment ». Comment va effectivement s'écrire le corps de cette fonction « plein » ? Nous avons esquissé l'algorithme dans le transparent précédent,

notes

résumé

6m 1s

Fonction plein

```
// plein(grille)
bool plein(const Grille& grille)
{
  // Si on trouve une case vide sur la premiere ligne, la gril
```

à savoir que si on trouve une case vide sur la première ligne, alors nous savons que la grille n'est pas encore remplie.

notes

résumé

6m 20s

Et c'est ce que nous allons concrètement coder. Nous allons itérer sur chacune des cases de la première ligne de la grille, « grille[0] », et tester. Si la case est vide, alors nous savons que la grille n'est pas pleine et c'est ce que la fonction « plein » va retourner. Sinon, eh bien naturellement, la grille est pleine, et à ce moment-là, nous retournons la valeur « true », pour indiquer que la grille est pleine. Voilà, nous avons ainsi terminé le codage de notre petite application du « Puissance 4 ». Le but de cette étude de cas était de vous montrer comment on aborde la programmation d'une telle application lorsque l'on débute en programmation. Plusieurs idées clés à retenir. La première est : toujours procéder de bas en haut. Commencer par identifier les types essentiels, les fonctionnalités fondamentales, avant d'affiner. Chaque fonction fondamentale peut elle-même à son tour, être modularisée, faire appel à d'autres fonctions ou plus de détails, qui vont l'aider à réaliser les traitements. Pour chacune des fonctions, toujours procéder avec méthodologie. Se poser la question du « quoi ». Quel est le rôle de la fonction dans le programme ? De quelles informations elle a besoin pour pouvoir fonctionner ? Qu'est-ce qu'elle va livrer comme information en retour ? Ensuite, se poser la question du « comment » et réfléchir à l'algorithme qui va l'aider à réaliser les traitements souhaités. Enfin, se préoccuper des aspects essentiels avant de passer à des aspects plus de cosmétiques, de peaufinage. Et dernier point essentiel : être systématique dans le test des différentes fonctionnalités, au fur et à mesure que l'on progresse dans la programmation de l'application. Voilà c'était un certain nombre de points clés essentiels que nous souhaitions illustrer au travers de cette étude de cas et qui, nous l'espérons, vous seront utiles lorsque vous aborderez par vous-même, la programmation de telles applications. la programmation de

notes

résumé

6m 26s

telles applications.

notes

résumé