Titre de l'article

Ghosts In The Stack
http://www.ghostsinthestack.org

TranceFusion

Résumé

Cet article introduit le concept de récursivité, technique couramment utilisé en programmation. Elle permet de résoudre certains problèmes de façon assez simple, comme par exemple des tris ou des calculs de suites, mais possède a des inconvénients.

Table des matières

1 Principe et exemples **1**

2 Cas général et cas d'arrêt **3**

3 Inconvénients **4**
3.1 Lenteur 4
3.2 L'oubli d'un cas d'arrêt est fatal 4

Introduction

Je vais décrire dans cet article un concept important dans la programmation (en C comme dans les autres langages) : la récursivité. C'est une méthode de programmation fréquemment employée dans la conception de programmes. Nous verrons qu'elle permet de faciliter l'écriture d'algorithmes et remplace les boucles itératives, mais elle possède un défaut qui est lié à la gestion de la mémoire et qui peut provoquer des erreurs si le programmeur estime mal le coût en calcul de son algorithme. J'en profite pour dire que quelques bases en mathématiques sont les bienvenues pour la lecture de cet article, car la notion de récursivité est fortement liée à la récurrence, par exemple pour le calcul des suites.

1 Principe et exemples

Qu'est-ce donc que la récursivité ? Il est difficile de donner une définition assez générale et formelle du mot "récursivité". Pour fixer les idées, nous allons voir que la récursivité appliquées aux fonctions : les fonctions récursives.

Définition : Une fonction récursive est une fonction qui s'appelle elle-même.

Plus généralement, on dit qu'une structure est récursive si elle contient un lien vers elle-même.

Le fait qu'une fonction peut s'appeler à l'intérieur d'elle même alors que sa définition n'est pas terminée peut choquer à première vue. Mais cela existe et est possible, voire même utile.

Si vous avez compris la définition, vous en avez certainement déduit que les fonctions récursives permettent de se passer des boucles itératives telles que for() et while(). Leur principal avantage est que quelquefois, on n'arrive pas à obtenir une solution itérative. C'est le cas par exemple des suites numériques récurrentes qui sont relativement complexes.

L'exemple bateau est celui de la fonction factorielle. La factorielle d'un nombre entier est le produit de tous les entiers strictement positifs et inférieurs ou égaux à ce nombre. Elle se note à l'aide d'un point d'exclammation situé après le nombre. Par exemple, la factorielle de 5 est $5! = 5*4*3*2*1 = 120$. Enfin, on note $0! = 1$ par convention (cela veut dire que la factorielle de zéro est égale à un). Voyons en C ce que cela donne :

```
int factorielle(int n)
{
  if (n == 0)
  {
    return 1;
  }
  else
  {
    return ( n * factorielle(n-1) );
  }
}


int main()
{
  printf("4! = %d",factorielle(4));
  getch();

  return 0;
}
```

La fonction factorielle() est une fonction récursive, comme vous l'aurez compris, car elle contient un appel à elle-même : factorielle(). Décomposons l'algorithme :
Si n = 0 , alors on retourne 1. C'est normal, car $0! = 1$ par convention.
Sinon, alors on retourne n multiplié par le résultat de la fonction factorielle avec n-1 en argument.

En clair, pour calculer la factorielle de 4, le programme éxécute ces appels :

```
factorielle(4) = 4 * factorielle(3)
               = 4 * 3 * factorielle(2)
               = 4 * 3 * 2 * factorielle(1)
               = 4 * 3 * 2 * 1 * factorielle(0)
               = 4 * 3 * 2 * 1 * 1
               = 24
```

Vous constaterez qu'à chaque appel, n est décrémenté de 1 puisqu'on multiplie le résultat par n. Les matheux auront compris que notre programme repose sur la propriété suivante qui peut servir de définition pour la factorielle :

n ! = n * (n-1) ! si n différent de 0, et n ! = 1 si n = 0

Ceci se démontre par récurrence à l'aide de la véritable définition de la factorielle qui est celle que je vous ai donné (n ! = produit des i pour i = 1 à n) . La démonstration étant vraiment triviale, je vous l'épargne...

Voici l'équivalent de la fonction factorielle() mais cette fois-ci en itératif, sans utiliser de récursivité.

```
int factorielle(int n)
{
  int result = 1;
  int i;
  for(i = 1; i<= n; i++)
  {
    result = result * i;
  }
  return result;
}
```

Ici, on utilise une variable result qui va contenir le résultat (variable de cumul), et un compteur i qui débute à 1 et est incrémenté jusqu'à n. A chaque itération, result est multiplié par i. On a donc bien le même shéma que tout à l'heure.

Notez que l'on aurait pû prendre pour exemple la fonction puissance(i, n) qui retourne iñ de manière récursive tel que iñ = i * i$\hat{}$(n-1) . La démonstration coule également de source. Je vous conseille de faire cette fonction d'une part en utilisant la récursivité (à l'aide de la formule que je viens juste de vous donner), d'autre part en utilisant les itérations en vous basant sur la définition iñ = produit de i pour j = 1 à i.

2 Cas général et cas d'arrêt

Nous allons généraliser le cas précédent sur la factorielle. Une fonction récursive doit TOUJOURS être composée des éléments suivants :

- un ou plusieurs "cas d'arrêt(s)". Ce sont des tests (souvent if()) vérifiant si le paramètre est égal à une valeur particulière. Si c'est le cas, alors la fonction s'arrête en retournant une valeur constante ; il n'y a pas d'appel récursif.
- un ou plusieurs cas généraux. La fonction éxécute ce(s) cas si le test du cas d'arrêt a été négatif. S'il y a plusieurs cas généraux, un autre test détermine lequel utiliser. Les cas généraux sont les cas dont la valeur du paramètre n'importe pas, et ils conduisent à un appel récursif.

Voilà, la définition formelle est donnée, maintenant tentons de mieux comprendre à travers l'exemple précédent. Dans la fonction factorielle, le cas d'arrêt est (n == 0) et conduit à la valeur constante 1. Le cas général est éxécuté si n!= 0 (comme dit plus haut, c'est l'inverse du cas d'arrêt) et est : "return (n * factorielle(n-1))".

Si tout ne semble pas très clair, comprenez simplement que cas d'arrêt signifie "valeur particulière de n pour laquelle la fonction doit s'arréter", et cas général signifie "appel récursif si n ne satisfait pas la (ou les) condition(s) du (ou des) cas d'arrêt".

J'ai dit plus haut qu'une fonction devait impérativement être composée d'au moins un cas d'arrêt et un cas général. Si vous réfléchissez, vous comprendrez vite pourquoi. Si l'on ne met pas de cas d'arrêt, la fonction ne s'interrompra jamais et la récursivité continuera "infiniment" (conduisant à un gros problème, voir plus bas). Si l'on ne met pas de cas général, il n'y a pas de récursivité ; il n'y a qu'un seul appel à la fonction, et elle n'est donc plus récursive.

Si vous souhaitez écrire une fonction récursive, je vous conseille de procéder comme ceci : commencez par chercher sur papier l'algorithme mathématique nécessaire pour votre fonction. Tentez de voir les valeurs particulières de vos paramètres qui entraînent l'arrêt de votre fonction (cas d'arrêt) , et d'autre part les instructions à éxécuter pour les autres valeurs (cas général). Je vous déconseille de vous lancer dirrectement dans la programmation sans étude au préalable, c'est le meilleur moyen pour se planter (à moins d'être à l'aise avec ces notions).

Si vous êtes matheux, vous comprendrez l'avantage des fonctions récursives notemment sur les suites numériques. Si vous avez une suite U(n) définie par récurrence, rien ne vous empêche de créer une foncion en C qui s'appellera int u(int n) et qui sera réursive ! Ensuite, vous pourrez tracer ses variation avec une librairie graphique de C... Vous constaterez la puissance de la récursivité, d'autant plus que quelquefois, la solution itérative est impossible. Il existe par exemple certaines suites définies par récurrence et il a été démontré qu'on ne peut pas trouver d'expression itérative de celles-ci...

Notez que vos fonctions récursives peuvent très bien contenir plusieurs (n, par exmeple) appels à elles-même, on parlera alors de **fonctions récursives d'ordre n**. Par exemple, certains algorithmes de tris sur des tableaux, dont le *tri rapide* utilisent la récursivité. Nous n'allons pas voir en détails l'algorithme du tri rapide ; pour plus d'infos, Wikipédia a un très bon article sur le sujet.

Mais les applications de la récursivité sont encore plus étendues que cela. Il est également possible de faire des [b]fonctions récursives croisées[/b] : considérez une fonction A et une fonction B telles que A appelle B et B appelle A. C'est tout à fait faisable en C, à moins d'avoir déclaré en tête du fichier source le prototype de la deuxième fonction (sinon le compilateur renverra une erreur). Ce genre de fonctions est plus difficile à concevoir, car il faut bien faire attention à prendre en compte tous les cas.

3 Inconvénients

3.1 Lenteur

D'une manière générale, on s'accorde à dire qu'une fonction récursive est plus lente que son équivalent en itératif. En effet, si l'algorithme est le même, le processeur a plus de tâches à exécuter si la fonction est récursive : il soit empiler à chaque appel l'adresse de retour, ainsi que les paramètres de la fonction, et les dépiler quand elle est finie. Tout cela prend du temps.

Cependant, certains compilateurs parviennent parfois à transformer des fonctions récursives pas trop compliquées en itérations, lors de la compilation, en fonction de l'optimisation demandée. Ainsi, la rapidité d'exécution s'en retrouve améliorée.

3.2 L'oubli d'un cas d'arrêt est fatal

Les fonctions récursives peuvent être très pratiques, comme nous l'avons vu. Mais elles peuvent aussi se révéler désastreuses si on en fait un mauvais usage. Voyons un premier exemple de fonction récursive vérolée :

```
void fonction()
{
```

```
  fonction();
}


int main()
{
  fonction();
  return 0;
}
```

Bouh ! C'est mal, il n'y a pas de cas d'arrêt ! Je vous avais dit que c'était dangereux, vous allez en avoir la preuve. Les boucles for() et while() peuvent engendrer le gel des programme à case d'une boucle infinie, mais les fonctions récursives sont plus vicieuses que cela...

Compilez, éxécutez et admirez. Le programme quitte moins d'une demi seconde après s'être lancé. Selon les OS, vous aurez même le droit à un beau message d'erreur. Maintenant, je vous pose la question : pourquoi ?

Mettez-vous à la place du système : une fonction, à peine déclarée, s'éxécute elle-même puis quitte. Autrement dit, elle se lance, et dès son lancement elle se relance, et la nouvelle fonction en relance une nouvelle, etc de manière infinie. En réalité, ce n'est pas tout à fait cela... Utilisons nos connaissances pour comprendre plus en détail ce qui se passe.

Lorsqu'un programme appelle une fonction, il saute au code de la fonction pour l'éxécuter (lors du call). Mais il doit pouvoir revenir là où il en était avant l'éxécution de la fonction. C'est pourquoi, l'adresse de retour de la fonction est stockée dans un endroit de la mémoire : la pile. Lors du "return" ou de la fonction, le programme récupère ("dépile") l'adresse de retour et saute à cette adresse.

Une fonction récursive est par définition une fonction donc elle stocke aussi l'adresse de retour sur la pile. Mais comme elle s'appelle elle-même, l'adresse de retour du 2ème appel va être stocké à la suite du premier. Et ainsi de suite... Si une fonction récursive comporte trop d'appels, toutes ces données s'empilent continuellement jusqu'à ce qu'on arrive alors au bout de la pile, et on a un dépassement de pile. Attention, c'est différent des buffers overflows où il y a juste débordement d'un tampon ; ici, on déborde carrément de la pile. On nomme également ceci une "explosion de pile".

Vous aurez compris que ce genre de fonction est dangereux pour la mémoire. Pour vous en convaincre, utilisez un débugger sur le programme et regardez l'état de la pile... Vous constaterez que sa fin est complétement écrasée d'adresse de retour pointant vers le main.

Ce type de fonction peut être exploitée pour un attaquant afin de ralentir une machine ou d'effectuer un DoS (déni de service) sur celle-ci. Nous venons de voir une fonction récursive linéaire. "Linéaire" ou "d'ordre 1" car la fonction ne s'appelle qu'une seule fois à chaque fois. Mais imaginez cette fonction :

```
void fonction()
{
  fonction();
  fonction();
}
```

"Ouais, on a juste deux fois plus d'appels, c'est tout", dites-vous ? Eh bien détrompez-vous ! On n'en a pas le double, mais le carré ! Eh oui, une fonction récursive d'ordre n comporte nî fonctions lancées en mémoire. Enfin nî fonctions sur la même "couche d'éxécution" de niveau i, mais en réalité, il y a (somme des nî pour i allant de 0 à n-1) fonctions lancées en tout en mémoire au bout de la n-ième série d'appels.

Une fonction récursive d'ordre n >= 1 a son nombre d'appels qui a une croissance exponentielle, c'est à dire qu'il ne croit pas constamment, mais que son taux de croissance augmente de plus en plus ! Plus il augmente, plus il augmente vite.

Enfin, pour en revenir à notre fonction récursive d'ordre 2 ou plus généralement d'ordre n, sachez que la puissance du DoS qui sera éxécuté va etre décuplée d'une facon phénoménale si n est grand. La pile explosera beaucoup plus tôt avec une telle fonction...

Moralité : Vérifiez toujours vos cas d'arrêts dans vos fonctions récursives. Regardez bien s'ils ont une chance d'être atteint pour éviter les problèmes. Et essayz d'évaluer le nombre maximal ou au moins le nombre moyen d'appels qui sera effectué par votre fonction en limitant le paramètre à une valeur maximale.

Conclusion

Nous avons vu que les fonctions récursives constituent un outil très puissant dans le calcul et la gestion des algorithmes récurrents. Là où les boucles classiques for() et while() sont plsu dur à implémenter, la récursivité est une solution simple. Mais une mauvaise maîtrise de cet outil peut conduire à des catastrophes au niveau de la mémoire. Cela demande donc une grande attention de la part du programmeur.

Et un petit conseil : quand vous le pouvez, évitez les fonctions récursives croisées, car cela devient vite le bazar à gérer. Ne perdez pas non plus de vue le fait que les fonctions récursives s'éxécutent entièrement à chaque appel, donc multiplier les tests (avec des if() redondants, par exemple) ne peut que les ralentir voire même embrouiller l'esprit du programmeur. Alors d'une manière plus simple, faites 2 fonctions simples au lieu d'une compliquée, même si l'une appelle l'autre qu'une fois.

Enfin, quand vous avez le choix entre solution itérative et récursive, préférez toujours la solution itérative !