Les fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle-même. Cela est donc équivalent aux boucles et permettent de répéter une suite d'instructions.

Dans un code, il suffit donc de mettre un appel vers une fonction dans le corps de cette fonction.

```
void f() {
    f();  // appel récursif de la fonction f
}
```

Même si ce n'est pas à proprement parlé des fonctions récursives, il est possible de considérer aussi les fonctions qui ne s'appellent pas elle-même directement, via d'autres fonctions intermédiaires, et qui forment des boucles d'appels de fonctions (récursivité croisée). Par exemple :

```
void g();  // déclaration anticipée

void f() {
    g();
}

void g() {
    f();
}
```

Dans ce code, la fonction `f` appelle la fonction `g`, qui appelle la fonction `f`, qui appelle la fonction `g`, et ainsi de suite.

Déclaration anticipée
Si vous lisez le code ligne par ligne, comme le fait le compilateur, vous remarquez que dans la fonction `f`, la fonction `g` n'est pas encore connue

par le compilateur. Vous aurez donc un message d'erreur “g n'est pas déclarée” :

```
main.cpp: In function 'void f()':
main.cpp:2:7: error: 'g' was not declared in this scope
     g();
       ^
```

Si vous définissez la fonction `g` avant la fonction `f`, vous aurez le même probleme inverse (“f n'est pas connu”).

Une déclaration anticipée permet de dire au compilateur que la fonction `g` existe, mais qu'elle sera définie plus tard. Et c'est suffisant pour le compilateur.

Une déclaration anticipée est simplement la signature d'une fonction, avec le corps de la fonction qui est remplacée par un point-virgule.

Si vous n'écrivez que la déclaration anticipée et oubliez de définir la fonction, vous aurez une erreur de lien (“référence indéfinie”) :

```
In function `f()': undefined reference to `g()'
```

Condition d’arrêt et continuation

Comme pour les boucles classiques, il est important de faire attention aux conditions d'arrêt, pour éviter les boucles infinies. Dans les codes précédents, il n'y a pas de condition d'arrêt, ce qui implique que ces boucles tourneront à l'infini, jusqu'à ce que le programme plante.

Contrairement aux instructions itératives vues précédemment, il n'y a pas de syntaxe explicites dans une fonction récursive pour arrêter la récursion. Vous devez donc utiliser un test conditionnel pour tester la condition d'arrêt et choisir de continuer ou non la récursivité.

Classiquement, cela veut dire que vous devez utiliser un test `if`, avec une condition d'arrêt associé à une clause `return` pour terminer la fonction, ou avec une condition de continuation associé à un appel

récursif de la fonction.

Par exemple, avec une condition d'arrêt :

```
void f() {
    if (CONDITION_ARRET) {
        return;
    }
    f();  // appel récursif
}
```

Dans ce code, lorsque la condition est vrai, l'instruction `return` est exécutée, ce qui termine la fonction `f`. Dans le cas contraire, la fonction `f` est appelée récursivement.

Un autre exemple, avec une condition de continuation :

```
void f() {
    if (CONDITION_CONTINUATION) {
        f();  // appel récursif
    }
}
```

Dans ce code, lorsque la condition est vraie, la fonction `f` est appelée récursivement. Dans le cas contraire, la fonction s'arrête.

Ce type d'approche est régulièrement utilisé en informatique, puisque bien adaptée pour certains types de problèmes : la recherche et le tri de collection par exemple.

Ce principe de récursivité peut être étendu également aux structures de données. Il existe en effet des structures de données conçues de façon récursives et l'utilisation d'algorithmes récursifs est particulièrement adapté dans ce cas. Un exemple que vous connaissez déjà est la collection associative `std::map`, qui utilise en interne un structure en arbre. (Les structures de données récursives seront étudiées dans la partie sur la programmation orientée objet).

Problématiques

La récursivité n'est pas une fonctionnalité triviale, il est important de bien comprendre les limitations avant de l'utiliser.

Pour commencer, selon la complexité des algorithmes, il n'est pas toujours simple de garantir qu'un algorithme récursif se termine correctement, quelques soit les conditions. Cela nécessite généralement plus de travail pour mettre en place un tel algorithme. La démonstration qu'un algorithme est correct peut être rapproché des raisonnement par récurrence en mathématique.

Le deuxième probleme est lié a la gestion des appels de fonction dans un ordinateur. Généralement, chaque appel de fonction sera ajouté à la Pile d'appel des fonctions : le contexte d’exécution de la fonction appelante sera sauvegardé et un nouveau contexte pour la fonction appelée est créé. Lorsqu'une fonction se termine, le contexte de la fonction appelée est détruit et le contexte de la fonction appelante est restauré. Tout cela est coûteux pour les performances (et un peu pour la mémoire).

Dans certaines conditions, le compilateur est capable de supprimer un appel de fonction, comme si il n'y avait pas d'appel de fonction du tout. Avec les fonctions récursives, cette optimisation est plus complexe pour le compilateur et il ne sera généralement pas capable de la faire.

Et ce n'est pas la seule optimisation que le compilateur sera incapable de faire. Plus généralement, le compilateur arrivera plus facilement à optimiser une boucle classique qu'un algorithme récursif. Par exemple pour dérouler une boucle (remplacer une boucle par une série d'instructions sans boucle) ou utiliser certaines fonctionnalités des processeurs (par exemple la parallélisation, pour exécuter plusieurs instructions en même temps).

Ces problématiques ne doivent pas vous empêcher d'utiliser la récursivité, mais simplement de bien faire attention avant de l'utiliser.

Un compteur récursif

Pour être concret, voici un exemple simple de fonction récursive, pour réaliser un compteur.

```
#include <iostream>

void f(int i) {
    std::cout << i << std::endl;
    if (i > 0) {
        f(i - 1);
    }
}

int main() {
    f(5);
}
```

affiche

```
5
4
3
2
1
0
```

Dans ce code, la fonction `f` prend en paramètre un compteur entier. Ce paramètre est passé par valeur, ce qui signifie que la valeur sera copiée lors de chaque appel à la fonction `f`.

La valeur du compteur est affichée dans un premier temps avec `std::cout`. Ensuite, cette valeur est testée : si elle est supérieur à zéro, la fonction `f` est appelée récursivement. Dans le cas contraire, la fonction s'arrête.

Le point important à noter est que le compteur est décrémenté à chaque appel récursif à la fonction `f`. Pour bien comprendre ce qui se passe, il faut “dérouler” manuellement la boucle :

- la fonction `main` appelle la fonction `f` avec la valeur `5` en argument ;
- dans le premier appel à la fonction `f`, le test `if` est vrai (5 est superieur à 0), la fonction `f` est appelée récursivement, avec la valeur 5 - 1 = 4 ;
- dans le deuxième appel à la fonction `f`, le test `if` est vrai (4 est superieur à 0), la fonction `f` est appelée récursivement, avec la valeur 4 - 1 = 3 ;
- dans le troisième appel à la fonction `f`, le test `if` est vrai (3 est superieur à 0), la fonction `f` est appelée récursivement, avec la valeur 3 - 1 = 2 ;
- dans la quatrième appel à la fonction `f`, le test `if` est vrai (2 est superieur à 0), la fonction `f` est appelée récursivement, avec la valeur 2 - 1 = 1 ;
- dans la quatrième appel à la fonction `f`, le test `if` est vrai (1 est superieur à 0), la fonction `f` est appelée récursivement, avec la valeur 1 - 1 = 0 ;
- dans la quatrième appel à la fonction `f`, le test `if` est faux (0 n'est pas superieur à 0), la fonction `f` se termine.

Diviser pour régner

“Diviser pour régner” (*divide and conquer* en anglais) est une stratégie de résolution de problématiques, basée sur la récursivité. Le principe générale consiste à prendre un problème complexe et de le diviser en sous-problèmes moins complexes. Puis de diviser encore ces sous-problèmes en sous-problèmes moins complexes, et ainsi de suite, jusqu'à obtenir des problèmes simples à résoudre.

Cette stratégie n'est pas spécifique à la programmation et il est assez facile de trouver un exemple dans la vie de tous les jours pour illustrer ce principe.

Imaginez par exemple que vous êtes chef de la police d'une ville et que vous devez trouver un voleur qui se cache dans votre ville. Vous ne savez

pour où il est, mais vous savez qu'il est dans la ville. Vous n'allez pas fouiller toutes les maisons vous-même, il vous faut de l'aide. Et vous ne pouvez pas simplement dire à vos policiers de fouiller les maisons au hasard, sinon certaines maisons seront fouillées deux fois et d'autres maisons ne seront pas fouillées : il vous faut vous organiser.

Vous allez commencer par diviser la ville en secteurs de recherche, par exemple par quartier. Et vous allez donner la responsabilité de fouiller chaque quartier à un inspecteur de police. Vous êtes donc sur que tous les quartiers seront fouillées une et une seule fois.

De la même façon, chaque inspecteur ne va pas fouiller chaque maison tout seul. Il va diviser sa zone de recherche en secteurs, par exemple par rues. Il va donc assigner un sergent de police à chaque rue, ce qui lui donne la garantie que toutes les rues seront fouillées et qu'aucune ne sera oubliée.

Pour terminer, chaque sergent va procéder de la même manière et va diviser la rue en zones de recherche plus petites : les maisons.

Cette division du travail complexe (fouiller une ville) en tâches de plus en plus simples (fouiller les quartiers, les rues puis les maisons) facilite l'organisation du travail et est assez naturelle à mettre en place pour cette problématique.

Un pseudo-code C++ équivalent pourrait ressembler au code suivant :

```
void fouiller(ZoneGéographique zone) {
    std::vector<ZoneGéographique > sousZones = diviser(zone);
    for (auto sousZone: sousZones) {
        fouiller(sousZone);  // récursion
    }
}

int main() {
    ZoneGéographique ma_ville;
    fouiller(ma_ville);
}
```

Ce code est quasiment une transcription littérale de la procédure

précédente, il est assez simple de comprendre le principe.

Transformation en itération

Comme expliqué au début de ce chapitre, les algorithmes récursifs peuvent être assimilés à des boucles. Il est donc naturel de se poser la question de savoir s'il est possible de transformer un algorithme récursif en algorithme itératif (utilisant une boucle classique).

Avec l'exemple précédent de la fouille d'une ville, il est possible de répartir le travail entre les policiers de façon différente. Il est par exemple possible de compter le nombre de rues dans la ville et le nombre de policiers disponibles, puis de calculer le nombre moyen de policiers par rue. Il est alors facile d'assigner les policiers pour chaque rue, avec une boucle classique.

Le pseudo-code C++ devient donc :

```
int main() {
    ZoneGéographique ma_ville;

    int nombreRues = compterRues(ma_ville);
    int nombrepoliciers = compterPoliciers(ma_ville);

    int tailleGroupes = nombrepoliciers / nombreRues;
    std::vector<Groupe> groupesPoliciers =
créerGroupesPoliciers(tailleGroupes);

    for (auto groupePolicier: groupesPoliciers) {
        fouiller(groupePolicier, rue);
    }
}
```

Dans ce code, la fonction récursive est remplacée par une boucle `for`.

Malgré le fait qu'il est possible de transformer une récursion en itération, cela ne retire par l'intérêt de la récursion dans certains. Il est possible d'imaginer que le nombre de maisons dans la ville ne soit pas le même dans chaque rue. Avec la réparation décrite ci-dessus, cela implique qu'il

y aura autant de policiers dans une rue qui contient 10 maisons, qu'une rue qui en contient 100 : la réparation du travail n'est pas optimale.

Dans la récursion, il est possible d'évaluer lors de chaque appel s'il reste beaucoup de travail à réaliser et choisir s'il est intéressant de continuer ou non la récursion.

Exercices

Aide : https://en.wikipedia.org/wiki/Recursion_(computer_science) et https://fr.wikipedia.org/wiki/Algorithme_r%C3%A9cursif

1. Ecrire une fonction récursive qui calcul la fonctionnelle d'un nombre entier : n! = 1 * 2 * 3 * ... * n.

2. Ecrire une fonction récursive qui calcul le plus grand commun diviseur (PGCD) de deux nombres entiers. Le PGCD est le plus grand entier qui divise ces deux nombres simultanément.

3. https://fr.wikipedia.org/wiki/Partition_d%27un_entier

4. Résoudre le jeu “les tours de Hanoi” par une fonction récursive. https://fr.wikipedia.org/wiki/Tours_de_Hano%C3%AF

5. recherche dichotomique.

Chapitre précédent	Sommaire principal	Chapitre suivant