

Récurtivité en Langage C

par [Franck.H](#)

Date de publication : 03/10/2006

Dernière mise à jour :

Etude des fonctions récursives en Langage C.

- I - Introduction
- II - Fonctions récursives
- III - Fonctions récursives terminales
- IV - Dangers et précautions
 - IV-A - Dépassement de capacité
 - IV-B - Débordement de pile (Stack Overflow)
 - IV-C - Forme itérative
- V - Code source complet avec exemple
- VI - Remerciements

I - Introduction

La notion de récursivité est avant tout un problème algorithmique plus qu'au niveau du langage lui même. Que ce soit en C, C++, Java, VB, Python, etc..., l'implémentation d'une fonction récursive se fera toujours plus ou moins de la même manière. Ici nous allons traiter de la récursivité avec le Langage C, telle est notre rubrique !

Mais qu'est-ce que la récursivité ? Et bien en fait d'un point de vue théorique cela reste assez simple ; il s'agit de programmes ou de fonctions d'un programme qui ont la faculté de s'appeler eux-mêmes (on entend également le terme *d'auto-appel* ce qui est logique). La récursivité est une manière simple et élégante de résoudre certains problèmes algorithmiques, notamment en mathématique mais cela ne s'improvise pas, il convient donc de savoir comment ce principe fonctionne.

Nous allons voir deux types de fonctions récursives: les *fonctions récursives* et les *fonctions récursives terminales*.

II - Fonctions récursives

Les fonctions récursives comme cité plus haut, sont donc des fonctions s'appelant elles-mêmes, elles sont également un moyen rapide pour poser certains problèmes algorithmique ; nous allons voir en détails comment elles fonctionnent.

Prenons un problème simple mais auquel vous n'avez peut-être pas pensé à utiliser la récursivité: le calcul d'une factorielle. Considérons $n!$ (qui se lit: factorielle de n) comme étant la factorielle à calculer, nous aurons ceci: $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$. Dans cette situation, nous pouvons déjà déterminer notre règle de sortie de notre fonction récursive: la valeur 1 qui symbolise la fin de la récursion !

En effet, il faut une condition de sortie pour la fonction, mais il faut être très vigilant quant au choix de la condition, vous devrez être sûr qu'elle soit validée à un moment ou à un autre sinon c'est comme si vous créez une boucle infinie sans condition de sortie !

La règle de récursion que nous devons définir, est le calcul de la factorielle en elle même soit, si nous considérons notre exemple sur $6!$, cité précédemment, nous pouvons définir notre règle de cette manière: $n! = (n) (n-1) (n-2) \dots (1)$. Nous pouvons en déduire que nous allons faire des appels en décrémentant la valeur de n à chaque appel de la fonction jusqu'à ce que $n == 1$!

Vous me direz sûrement, quoi des maths alors que nous parlons de développement en Langage C ! Hé oui, comme cité plus haut c'est avant tout un problème algorithmique et notre exemple est un exemple mathématique comme c'est le cas pour beaucoup d'algorithmes tout de même mais je vous rassure, nous n'irons pas plus loin que ca en maths, c'est tout , nous avons ce qu'il nous faut pour créer notre fonction récursive, la voici:

Fonction récursive simple

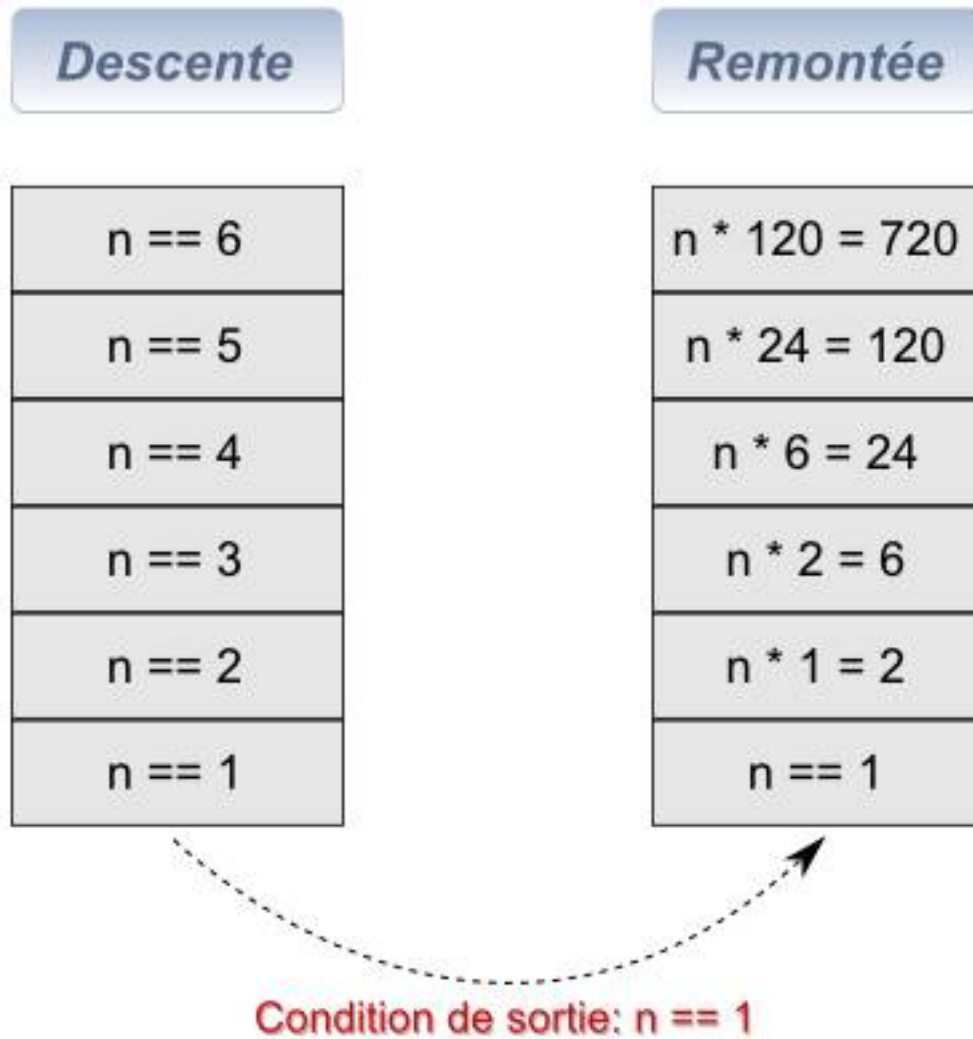
```
unsigned long factoriel (int n)
{
    if (n < 0) {
        exit (EXIT_FAILURE);
    }
    else if (n == 1 || n == 0) {
        return 1L;
    }
    return n * factoriel (n - 1);
}
```

Nous pouvons observer ici que le dernier **return** est en fait l'appel récursif et nous soustrayons 1 à chaque appel jusqu'à ce que $n == 1$ qui est, comme décrit plus haut, notre condition de sortie.

Non, cela ne s'arrête pas là et c'est ici que nous allons voir le fonctionnement des fonctions récursives. Lorsque la fonction rencontre la condition de sortie, elle remonte dans tous les appels précédents pour calculer n avec la valeur précédemment trouvée !

Les appels des fonctions récursives sont en fait *empilées* (pile qui est une structure de donnée régie selon le mode *LIFO*: Last In First Out, Dernier Entré Premier Sorti). Chaque appel se trouve donc l'un à la suite de l'autre dans la pile du programme. Une fonction de ce type possède donc deux parcours: la *phase de descente* et la *phase de remontée*.

Voyons ceci grâce à un petit schéma:



Nous voyons très bien la phase de descente et de remontée dans la pile des appels de la fonction récursive. Ce n'est qu'au moment de la remontée, donc également au moment où la condition de sortie est vraie, que les appels enregistrés sont dépilés au fur et à mesure de la remontée. Ici pour des petits calculs cela convient très bien mais lorsqu'il s'agit de faire de plus profondes récursions, un autre type de fonction récursive existe mais n'est pas forcément connue de tout le monde, c'est la *récursivité terminale*, que nous allons étudier dans le prochain chapitre.

III - Fonctions récursives terminales

Avec ce que nous avons vu plus haut, imaginez un instant que vous devez calculer le factoriel d'un très grand nombre, comme par exemple, un calcul de probabilité sur le tirage des boules de Loto ! Oui, si vous connaissez les calculs de probabilités voir même les combinatoires, vous savez que les calculs de factoriels y sont omniprésents mais, je ne vais pas trop entrer dans les détails pour ceux qui ne sont pas très familiers avec ce genre de connaissances mathématiques.

Nous allons simplement imaginer les inconvénients qu'une récursivité normale peut avoir sur des récursions plus profondes. Imaginez une grille de Loto, elle contient 49 numéros dont seulement 6 peuvent être tirés au sort. En restant dans un calcul simple pour ne pas trop nous dérouter du sujet, nous aurions une formule comme ceci pour calculer des probabilités:

$$\binom{49}{6} = \frac{49!}{(49-6)! 6!}$$

Ici, nous devrions calculer le factoriel de 49 ce qui nous donne: $49! = 49 \times 48 \times 47 \dots 8 \times 7 \times 6!$; le résultat obtenu est d'une grandeur inimaginable !

Mais cela n'est pas le sujet, c'est juste pour vous montrer que de cette manière, avec une récursivité normale, nous avons $(49-6) \times 2$ passages soit 43 appels empilés l'un après l'autre sans compter qu'il faut également remonter tous les appels ce qui nous fait un total de 86 passages. Vous pouvez vous imaginer la perte de temps sur des récursions encore plus profondes et qui risqueraient par ailleurs de faire exploser la pile ce qui conduirait irrémédiablement au plantage du programme !

C'est là que peut intervenir les *récursions terminales* ! Mais c'est quoi ? Me direz-vous ! Et bien c'est une récursion avec uniquement une phase de descente, sans remontée. Ceci est possible car la dernière expression **return factoriel_terminale (...)** de notre fonction nous renvoie directement la valeur obtenue par l'appel récursif courant, sans qu'il n'y ait d'autres opérations à faire, ce qui n'est pas le cas dans notre fonction récursive simple, où l'on multiplie n par le retour de la fonction. En conséquence, les appels de la fonction n'ont pas besoin d'être empilés car l'appel suivant remplace simplement l'appel précédent dans le contexte d'exécution.

En réalité, il n'y a aucune garantie que le compilateur puisse optimiser le code pour avoir une récursivité terminale.

En considérant le même calcul que dans le chapitre précédent, soit calculer $6!$, voici l'implémentation de notre fonction récursive terminale:

Fonction récursive terminale

```
unsigned long factoriel_terminal (int n, unsigned long result)
{
    if (n < 0) {
        exit (EXIT_FAILURE);
    }

    if (n == 1) {
        return result;
    }
    else if (n == 0) {
        return 1L;
    }
}
```

Fonction récursive terminale

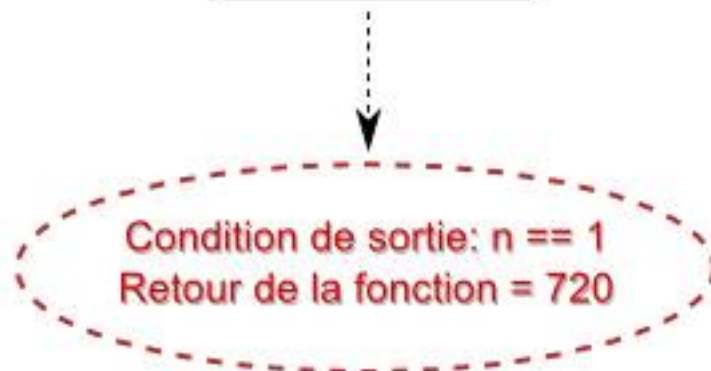
```
    return factoriel_terminal (n - 1, n * result);  
}
```

Nous remarquons ici que nous avons pris un argument supplémentaire, ceci est un passage obligatoire pour créer une récursivité terminale ; de cette manière, la dernière instruction est belle et bien la fin de l'appel courant de notre fonction et donc l'appel suivant peut prendre la place de la précédente car le résultat se trouve dans notre second argument.

Comme à l'accoutumée, observons un petit schéma pour mieux nous représenter le parcours de la fonction:

Descente

n == 6 result = 1
n == 5 result = 6
n == 4 result = 30
n == 3 result = 120
n == 2 result = 360
n == 1 result = 720



Le schéma nous prouve bel et bien qu'il n'y a qu'une phase de descente mais pas de remontée, De cette manière nous économisons l'utilisation de la pile du programme, et nous gagnons également du temps en exécution, c'est

prodigieux !

IV - Dangers et précautions

Les fonctions récursives étant un moyen assez puissant pour résoudre certains problèmes de façon élégante, elles n'en restent pas moins dangereuses et ce pour plusieurs raisons.

IV-A - Dépassement de capacité

Une des causes assez fréquente quand vous travaillez sur de très grands nombres, est le dépassement de capacité. C'est un phénomène qui se produit lorsque vous essayez de stocker un nombre plus grand que ce que peut contenir le type de votre variable.

Il est d'usage de choisir un type approprié, même si vous êtes certains que le type que vous avez choisi ne sera jamais dépassé, utilisez tant que possible une variable pouvant contenir de plus grandes données. Ceci s'applique à tous types de données.

Évitez le type **int** si vous travaillez avec une fonction récursive, comme les exemples précédents pour le calcul de *factoriels*. Ce type est très petit et dans une fonction récursive il peut très vite arriver de le dépasser et c'est donc le plantage assuré.

Préférez-lui un type comme **long** pour assurer un minimum la viabilité de votre application !

*Il faut noter que la taille d'un **int** peut être différente suivant les implémentations systèmes. En effet, par exemple en 32 bits avec un compilateur Microsoft (c), la taille d'un **int** est la même que celle d'un **long**, il est donc préférable de se renseigner sur la taille des variables suivant votre système !*

IV-B - Débordement de pile (Stack Overflow)

Ceci est sans doute une des causes les plus souvent rencontrés dans le plantage de programmes avec des fonctions récursives. Nous savons que les appels récursifs de fonctions sont placés dans la pile du programme, pile qui est d'une taille assez limitée car elle est fixée une fois pour toutes lors de la compilation du programme.

Dans la pile sont non seulement stockés les valeurs des variables de retour mais aussi les adresses des fonctions entre autres choses, les données sont nombreuses et un débordement de la pile peut très vite arriver ce qui provoque sans conteste une sortie anormale du programme.

Dans l'exemple de la fonction **factoriel**, il nous faut (en arrondissant) environ 135000 appels récursifs pour faire exploser la pile. Vous me direz, c'est déjà pas mal mais je ne jure de rien quand il s'agit d'applications scientifiques !

Une autre méthode existe cependant ! Si vous êtes presque sûr de dépasser ce genre de limites, préférez alors une approche itérative plutôt qu'une approche récursive du problème. Une approche récursive demande beaucoup de moyens en ressources car énormément de données doivent être stockées dans la pile d'exécution alors qu'en revanche, une approche itérative telle une boucle **for** est bien moins coûteuse en terme de ressources et est bien plus sûre, sauf dans le cas d'un dépassement de capacité bien sûr !

IV-C - Forme itérative

Voyons en vitesse une possibilité de forme itérative pour notre calcul de $6!$:

Forme itérative

```
unsigned long factoriel_iterative (unsigned int n)
{
    unsigned long ret = 1;
    unsigned int i = 1;

    for (i = 1; i <= n; i++)
    {
        ret *= i;
    }

    return ret;
}
```

Vous voyez que cela reste simple mais bien sûr, cette forme est un peu moins lisible qu'une approche récursive. L'avantage ici, réside dans le fait que vous ne risquez pas de débordement de pile mais très certainement un dépassement de capacité qui dans ce cas fait également terminer le programme en retournant le signal *SIGFPE* qui est une constante standard qui indique diverses opérations mathématiques incorrectes telle qu'une division par zéro ou dans notre cas un dépassement de capacité !

V - Code source complet avec exemple

Ici, vous pouvez télécharger le code source complet avec le programme d'exemple et un projet Code::Blocks version Windows: [recursivite.zip](#)

```
#include <stdio.h>
#include <stdlib.h>

/*
 * Fonction recursive simple.
 */
unsigned long factoriel (int n)
{
    if (n < 0) {
        exit (EXIT_FAILURE);
    }
    else if (n == 1 || n == 0) {
        return 1L;
    }

    return n * factoriel (n - 1);
}

/*
 * Fonction recursive terminale.
 */
unsigned long factoriel_terminal (int n, unsigned long result)
{
    if (n < 0) {
        exit (EXIT_FAILURE);
    }

    if (n == 1) {
        return result;
    }
    else if (n == 0) {
        return 1L;
    }

    return factoriel_terminal (n - 1, n * result);
}

/*
 * Fonction iterative.
 */
unsigned long factoriel_iterative (unsigned int n)
{
    unsigned long ret = 1;
    unsigned int i = 1;

    for (i = 1; i <= n; i++)
    {
        ret *= i;
    }

    return ret;
}

int main (void)
{
    unsigned long ret = 0;

    /*
     * Recursivite simple.
     */
    ret = factoriel (6);
    printf ("6! = %ld\n", ret);

    /*
```

```
    * Recursivite terminale.
    */
    ret = 0;
    ret = factoriel_terminal (6, 1);
    printf ("6! = %ld\n", ret);

    /*
    * Fonction iterative.
    */
    ret = 0;
    ret = factoriel_iterative (6);
    printf ("6! = %ld\n", ret);

    return EXIT_SUCCESS;
}
```

VI - Remerciements

Un grand merci à farscape, PRomu@ld, gl et gorgonite pour leurs avis, remarques, conseils et précisions. Egalement un gros merci à wichtounet pour sa relecture et correction !