



Centre Informatique pour les **L**ettres
et les **S**ciences **H**umaines

Apprendre C++ avec Qt : Leçon 15

Fonctions récursives

1 - Un appel récursif n'est pas forcément un cercle vicieux.....	2
2 - Objets communs à toutes les exécutions enchâssées	3
Fonctions membre récursives et variables membre (1er épisode)	4
Paramètres de type référence ou pointeur.....	5
3 - Objets propres à chaque exécution.....	5
Variables locales.....	5
Passage de valeurs	6
Fonctions membre récursives et variables membre (2° épisode)	7
4 - Valeur renvoyée par une fonction récursive	7
5 - Un appel récursif peut simplifier l'écriture d'un programme	8
6 - Bon, c'est gentil tout ça, mais ça fait quand même 7 pages. Qu'est-ce que je dois vraiment en retenir ?.....	8

Lorsqu'on la rencontre pour la première fois, l'idée selon laquelle les instructions placées à l'intérieur du corps d'une fonction pourraient comporter un appel à la fonction elle-même semble souvent inepte. Comment une suite d'instructions pourrait-elle décrire valablement la façon d'exécuter un traitement, si l'une de ces instructions exige l'application du traitement ? C'est un peu comme si les instructions d'assemblage d'un meuble livré en kit comportaient une étape enjoignant de... monter le meuble. En programmation, pourtant, ce phénomène est si fréquent qu'il porte un nom : les fonctions qui s'appellent elles-mêmes sont dites récursives.

1 - Un appel récursif n'est pas forcément un cercle vicieux

Autant le dire tout de suite : pratiqué sans discernement, un appel récursif EST un cercle vicieux. Un appel à la fonction suivante¹, par exemple, ne nous rendra jamais riches :

```
1 void devineResultatTierce(int & premier, int & second, int & troisieme)
2 {
3   devineResultatTierce(premier, second, troisieme); //appel récursif ?
4 }
```

Cette fonction revient simplement à dire que pour deviner le résultat du prochain tiercé, il suffit de deviner le résultat du prochain tiercé. La nouvelle exécution de la fonction déclenchée par l'appel récursif n'est pas plus avancée que la première. Tout ce qu'elle peut faire, c'est déclencher une troisième exécution, et ainsi de suite, à l'infini.

Ou, plus exactement, jusqu'à ce que la mémoire disponible soit épuisée. Lors d'un appel de fonction, il faut en effet que le processeur conserve une trace de l'état d'avancement de la fonction appelante, de façon à pouvoir continuer lorsque l'exécution de la fonction appelée sera achevée. Si minime que soit la quantité de mémoire nécessaire à cette conservation, quelques secondes suffiront certainement à la fonction ci-dessus pour déclencher les millions d'exécutions emboîtées les unes dans les autres qui, en exigeant que l'état d'avancement de chacune d'entre elles soit conservé, finiront par épuiser la mémoire disponible, quelle que soit la taille de celle-ci.

Pour que le programme ne sombre pas, victime d'une récursion sans fin, il faut que l'appel récursif ne soit exécuté que conditionnellement.

Examinons par exemple la fonction suivante :

```
1 void penible()
2 {
3   char reponse;
4   cout << "Voulez-vous continuer à vous enfoncer dans la récursion ?\n";
5   cin >> reponse;
6   if(reponse != 'N')
7     penible();           //appel récursif CONDITIONNEL !
8 }
```

Ce fragment de code, comme les autres exemples proposés dans cette Leçon, suppose un contexte d'entrées/sorties "consoles" dans lequel l'insertion dans `cout` (4) provoque un affichage à l'écran alors que l'extraction depuis `cin` (5) lit le clavier. Ce contexte (un peu archaïque, à vrai dire) présente l'avantage de permettre aux fonctions de produire une trace de leur exécution sans qu'aucune mise en place et transmission de variables ne soit nécessaire.

Il y a, dans ce cas, fort à parier que l'utilisateur finira par se lasser et par presser la touche libératrice, bien avant que la mémoire disponible n'ait été épuisée par la récursion. Que se passe-t-il alors ? L'exécution en cours ne procède à aucun appel récursif, mais s'achève normalement. L'exécution de la fonction appelante peut donc reprendre, ce qui signifie ici qu'elle peut elle-même s'achever. La résolution de la situation se propage ainsi jusqu'au niveau le plus élevé, où l'achèvement de l'exécution signifie l'achèvement de la récursion.

Une façon imagée de se représenter la situation réalisée lors de l'exécution d'une récursion est de remplacer mentalement l'appel récursif par une copie du code de la fonction.

Dès que les fonctions concernées cessent d'être très simples, une telle représentation ne peut plus vraiment être réalisée dans tous ses détails, mais son principe peut vous aider à ne pas perdre le fil, ou à le retrouver lorsque vous l'aurez perdu...

¹ Même une fonction comportant un appel récursif ne se mettra pas spontanément en route ! Il faut, comme toujours, qu'une autre fonction fasse un appel (non récursif, par définition) à la fonction pour que celle-ci soit exécutée.

Voici ce que donnerait cette représentation dans le cas où un utilisateur, confronté à notre fonction `penible()`, répondrait quatre fois 'O', avant de finir par dire 'N' :

```
{//CECI N'EST PAS UNE FONCTION, MAIS UNE REPRESENTATION D'UNE EXECUTION
char reponse;
cout << "Voulez-vous continuer à vous enfoncer dans la récursion ?\n";
cin >> reponse;
if(reponse != 'N')
    {//il a répondu O (le fou !)
    char reponse;
    cout << "Voulez-vous continuer à vous enfoncer dans la récursion ?\n";
    cin >> reponse;
    if(reponse != 'N')
        {//il a répondu O (le fou !)
        char reponse;
        cout << "Voulez-vous continuer à vous enfoncer dans la récursion ?\n";
        cin >> reponse;
        if(reponse != 'N')
            {//il a répondu O (le fou !)
            cout << "Voulez-vous continuer à vous enfoncer dans la récursion ?\n";
            cin >> reponse;
            if(reponse != 'N')
                // il a répondu N (enfin !)
            }
        }
    }
}
```

Cette représentation concrétise la notion d'enchâssement des exécutions : il est manifeste que l'exécution (bleue) du deuxième appel est à l'intérieur de l'exécution (noire) du premier, qui reste en suspens et ne s'achèvera que lorsque le deuxième appel se sera lui-même achevé. Comme la fin de l'exécution bleue dépend à son tour de l'achèvement de la rouge, qui lui-même exige la fin de la verte, qui n'est atteint que lorsque l'exécution de la mauve est terminée, on voit bien que la réponse 'N' fournie lors de l'exécution mauve provoque non seulement la fin de cette exécution mais, en cascade, celles de toutes les exécutions de niveau supérieur.

Remarquons au passage que cette façon de représenter l'exécution d'un appel de fonction peut très bien être appliquée à un appel "ordinaire" (i.e. non récursif). Dans certains cas, il ne s'agit d'ailleurs pas d'une simple vue de l'esprit, comme vous pourrez le découvrir en lisant [l'Annexe 6 : Fonctions inline](#).

Si la notion de fonction récursive ne pose guère de problème², la maîtrise des appels récursifs exige de bien comprendre les conséquences de l'enchâssement des exécutions. Chacune de ses exécutions dispose en effet de son propre jeu d'objets locaux, et l'usage qui peut être fait des objets accessibles à une fonction récursive dépend donc fondamentalement de la réponse à la question suivante : toutes les exécutions enchâssées travaillent-elles sur les mêmes objets ?

2 - Objets communs à toutes les exécutions enchâssées

Les objets "communs" à toutes les exécutions établissent une communication bidirectionnelle entre les différents niveaux d'exécution : une exécution enchâssée dispose d'un objet qui est initialement dans l'état où l'exécution appelante l'a mis avant l'appel et, lorsque l'exécution enchâssée s'achève, l'exécution appelante reprend avec un objet dont l'état dépend des actions effectuées par l'exécution enchâssée.

Lorsque tous les objets manipulés par une fonction récursive sont "communs", l'effet de l'appel récursif reste assez proche de celui qu'aurait produit une simple itération obtenue à l'aide d'une quelconque structure de contrôle. Les deux exemples qui vont suivre sont d'ailleurs fonctionnellement équivalents à quelque chose comme :

```
do {
    cout << val << " ";
} while(--val > 0 )
```

² Il ne s'agit, après tout, qu'une fonction qui s'appelle elle-même. D'un certain point de vue, cette définition ne justifie sans doute pas une Leçon complète : elle tient parfaitement dans une petite note de bas de page !

Fonctions membre récursives et variables membre (1er épisode)

Lorsqu'une fonction membre comporte un **appel récursif** effectué (implicitement) au titre de l'instance utilisée pour l'appel initial, toutes les exécutions enchâssées partagent un unique jeu de variables membre. Une variable membre peut ainsi être utilisée pour contrôler la profondeur de la récursion, comme dans l'exemple suivant :

```

1 class CExemple
2 {
3 protected:
4     int compteur;
5 public:
6     CExemple(int val = -1) : compteur(val){}
7     void affiche();
8 };

```

Si la fonction `affiche()` est définie ainsi :

```

1 void CExemple::affiche()
2 {
3     cout << compteur << " ";
4     if (--compteur > 0)
5         affiche(); //appel récursif ne spécifiant explicitement aucune instance
6 }

```

l'exécution du code suivant

```

1 CExemple uneInstance(12);
2 uneInstance.affiche();

```

se traduira par l'affichage de

```
12 11 10 9 8 7 6 5 4 3 2 1
```

et par la remise à zéro du contenu de la variable `uneInstance.compteur`.

La fonction `affiche()` commence par afficher le contenu de la variable membre. Comme celle-ci a été initialisée à 12, c'est cette valeur qui apparaît. Lors de l'évaluation de **l'expression contrôlant l'appel récursif conditionnel**, la première opération effectuée est de décrémenter le compteur (puisque l'opérateur `--` est ici préfixé). La nouvelle valeur, 11, est ensuite comparée à 0 et, comme elle est supérieure, **l'appel récursif** est effectué. Comme il s'agit de l'appel d'une fonction membre et qu'aucune instance n'est spécifiée, cet appel est automatiquement effectué au titre de l'instance sur laquelle travaille la fonction appelante, c'est à dire `uneInstance`. L'exécution enchâssée ainsi déclenchée dispose donc d'un compteur qui ne contient plus que 11, et elle affiche cette valeur avant de procéder à son tour à la décrémentation du compteur et à un deuxième appel récursif. Au cours du 11^e appel récursif, le compteur ne contient plus que 1. Une fois cette valeur affichée, la décrémentation conduit à une valeur nulle, et cette exécution s'achève donc sans appel récursif. Cet achèvement permet au 10^e appel récursif de s'achever à son tour et, de proche en proche, tous les appels enchâssés se terminent, y compris **l'appel initial** (non récursif). Le compteur, pour sa part, n'est plus affecté et reste donc nul.

Le facteur déterminant est ici que l'appel récursif exploite le privilège des fonctions membre qui les autorise à accéder aux membres de leur classe sans préciser au titre de quelle instance elles veulent le faire. L'usage implicite du pointeur caché `this` (cf. [Leçon 9](#)) garantit que, par défaut, c'est l'instance au titre de laquelle la fonction a été appelée qui sera utilisée. Dans le cas de **notre appel récursif**, la fonction membre s'appelle donc elle-même au titre de l'instance qui a été utilisée lors de **l'appel initial** (non récursif).

Cet exemple illustre bien le fait que la variable membre établit une communication bidirectionnelle entre les différents niveaux d'exécution :

- Chaque exécution trouve dans la variable membre la valeur qu'y a placé l'exécution du niveau immédiatement supérieur (c'est ce qui permet à la récursion de s'achever). Ce type de communication pourrait être appelée *descendante*.
- Lorsqu'une exécution s'achève, celle qui l'a déclenchée retrouve la variable membre dans l'état où l'a laissée l'exécution qui vient de s'achever (c'est pour cela que la variable membre contient finalement zéro). Ce type de communication pourrait être appelée *ascendante*.

Paramètres de type référence ou pointeur

Si une fonction récursive reçoit comme paramètre une **référence** (ou l'adresse d'une variable), elle opère sur la variable qui (ou dont l'adresse) lui est transmise, et non sur une copie locale de celle-ci. Les exécutions enchâssées travailleront donc sur le même objet, comme dans le cas de la variable membre étudié ci-dessus. Il n'est donc pas étonnant de constater que, si la fonction `affiche()` devient :

```
1 void CExemple::affiche(int & p)
2 {
3     cout << p << " ";
4     if (--p > 0)
5         affiche(p); //appel récursif
6 }
```

son exécution par

```
1 int n = 12;
2 CExemple uneInstance; //affiche() n'utilise plus la variable membre
3 uneInstance.affiche(n);
```

conduit au même résultat que celui obtenu précédemment, c'est à dire la mise à zéro du contenu de la variable `n` et l'affichage suivant :

```
12 11 10 9 8 7 6 5 4 3 2 1
```

Bien entendu, le fait que la fonction récursive soit ou non membre d'une classe ne change absolument rien au fait que le passage d'une référence (ou d'un pointeur) rend l'objet concerné "commun" et établit donc une communication bidirectionnelle entre les différents niveaux.

Remarquez que, dans cet exemple, la fonction ne fait aucun usage d'un autre membre de la classe. Elle fonctionnerait donc exactement de la même façon si elle était globale, mis à part, évidemment, le fait que l'appel initial ne devrait alors pas être fait au titre d'une instance de `CExemple`.

3 - Objets propres à chaque exécution

Le cas des objets propres aux différents niveaux d'exécution est plus intéressant. En effet, lorsque les différents niveaux d'exécution d'une fonction récursive manipulent des objets différents, l'équivalence avec une formule purement itérative cesse d'être évidente.

En théorie, une fonction récursive peut toujours être remplacée par un codage purement itératif (et réciproquement). D'un point de vue pratique, toutefois, cette possibilité n'est pas toujours très attrayante : certains cas de figure semblent se prêter plus volontiers à une approche itérative, alors que d'autres sont plus facilement abordables en utilisant la récursion.

Variables locales

Par définition, les variables locales à une fonction sont recrées à chaque fois qu'une exécution de la fonction est lancée. Dans le cas d'un appel récursif, chaque exécution disposera donc de son propre jeu de variables locales, ce qui signifie que ces variables ne peuvent pas être utilisées pour contrôler la récursion ou, plus généralement, pour établir une communication entre les différents niveaux d'enchâssement³. En revanche, cette caractéristique des variables locales les protège contre les effets des appels enchâssés, ce qui explique que, si la fonction `affiche()` est privée de paramètre et définie ainsi :

³ Sauf, bien entendu, si l'appel récursif transmet une référence à l'une d'entre elles, ou un pointeur sur l'une d'entre elles. Du point de vue de la récursion, la variable locale concernée ressemble alors à une variable membre impliquée dans un appel récursif effectué (implicitement) au titre de `*this`.

```

1 void CExemple::affiche()
2 {
3     int locale = compteur;
4     if (--compteur > 0)
5         affiche(); //appel récursif
6     cout << locale << " ";
7 }

```

son appel par

```

1 CExemple uneInstance(12); //affiche() utilise à nouveau la variable membre
2 uneInstance.affiche();

```

affichera

```
1 2 3 4 5 6 7 8 9 10 11 12
```

Si ce résultat vous surprend, essayez de vous représenter l'enchâssement des appels. L'instruction d'affichage apparaissant après l'appel récursif, elle ne sera exécutée qu'après achèvement des exécutions enchâssées, au cours de ce qu'on appelle la *remontée récursive*. Le premier affichage effectivement réalisé est donc celui qui correspond à l'exécution la plus profondément enchâssée (qui est, par définition, la première à se terminer), ce qui explique que les valeurs apparaissent dans l'ordre croissant, à l'inverse des exemples précédents. Remarquez que, bien que la dernière valeur affichée soit 12, l'appel de la fonction `affiche()` se solde toujours par l'annulation du contenu du compteur.

Passage de valeurs

Lorsqu'une fonction reçoit une simple valeur destinée à initialiser un de ses paramètres, ce paramètre est un objet propre à la fonction, exactement comme dans le cas d'une variable locale. La version suivante de la fonction `affiche()` utilise cette propriété pour afficher deux fois la valeur qui lui est transmise : une fois **avant** de procéder à l'appel récursif, et une seconde fois **après**.

```

1 void CExemple::affiche(int p)
2 {
3     cout << p << " ";
4     if (p > 1)
5         affiche(p-1); //appel récursif
6     else
7         cout << "\nA la remontée : "; //on est arrivé au fond !
8     cout << p << " ";
9     p = 0; //cette instruction n'a aucun effet
10 }

```

Comme la valeur du paramètre est parfaitement protégée contre les effets des exécutions enchâssées, c'est bien la même valeur qui est affichée deux fois par chaque exécution. Toutefois, comme ces deux affichages sont séparés par un **appel récursif**, ils n'apparaissent pas l'un après l'autre à l'écran, mais sont séparés par les affichages produits par les exécutions enchâssées. Seule l'exécution la plus profonde (qui est, par définition, dispensée d'appel récursif) est en mesure d'afficher **tout ce qu'elle a à afficher** sans être interrompue. Elle en profite (7) pour passer à la ligne et annoncer le début de la remontée récursive.

Si cette fonction est appelée par

```

1 CExemple uneInstance; //affiche() n'utilise plus la variable membre
2 cout << "A la descente : ";
3 uneInstance.affiche(12);

```

on obtient l'affichage suivant :

```
A la descente : 12 11 10 9 8 7 6 5 4 3 2 1
A la remontée : 1 2 3 4 5 6 7 8 9 10 11 12
```

Dans cet exemple, la communication descendante rendue possible par le passage d'une valeur est utilisée à la fois pour afficher les différentes valeurs et pour garantir que la récursion n'est pas infinie. L'absence de communication ascendante est, pour sa part, mise en évidence par le fait que la mise à 0 du paramètre (9) ne perturbe aucunement l'affichage des valeurs lors de la remontée récursive.

Fonctions membre récursives et variables membre (2° épisode)

Lorsqu'une fonction membre fait un appel récursif, elle n'est bien entendu pas obligée de le faire au titre de l'instance pour laquelle elle a été appelée. Si l'appel récursif est fait au titre d'une nouvelle instance (une variable locale à la fonction, par exemple), les variables membre sur lesquelles travaillera par défaut la nouvelle exécution ainsi déclenchée seront évidemment celle de cette nouvelle instance.

Cette façon de procéder permet de contrôler la récursion sans utiliser de paramètre explicite⁴ et sans modifier le contenu de la variable membre. La fonction `affiche()` pourrait donc être définie comme ceci :

```

1 void CExemple::affiche()
2 {
3     cout << compteur << " ";
4     CExemple instanceLocale(compteur-1);
5     if (compteur > 1)
6         instanceLocale.affiche();
7     else
8         cout << "\nA la remontée : ";
9     cout << compteur << " ";
10 }
```

L'exécution du code suivant

```

1 CExemple uneInstance(12); //affiche() utilise à nouveau la variable membre
2 cout <<"A la descente : ";
3 uneInstance.affiche();
```

produirait alors le même affichage que celui obtenu avec la version précédente :

```

A la descente : 12 11 10 9 8 7 6 5 4 3 2 1
A la remontée : 1 2 3 4 5 6 7 8 9 10 11 12
```

La variable membre n'étant pas modifiée, l'affichage effectué lors de la remontée récursive est le même que celui effectué lors de la descente. Chaque exécution enchâssée s'effectue sur une instance dont le compteur a été initialisé avec une valeur inférieure d'une unité à celle du compteur de l'instance utilisée par l'exécution appelante. Cette initialisation permet à la fois l'affichage de valeurs différentes par les différents niveaux d'exécution et l'achèvement de la récursion.

4 - Valeur renvoyée par une fonction récursive

Lorsqu'un traitement doit être appliqué à une série de données, il est souvent pratique d'écrire une fonction n'effectuant le traitement que sur un élément et s'appelant elle-même pour effectuer le traitement sur la série amputée de l'élément déjà traité. Si le traitement produit un résultat (plutôt qu'un effet), il est nécessaire de cumuler les résultats obtenus par les exécutions enchâssées, et ce cumul est facilement obtenu en tirant parti du fait qu'il doit, de toute façon, constituer la valeur renvoyée par la fonction (de façon à ce que l'appel initial produise le résultat final) .

La fonction suivante applique ce principe au dénombrement des occurrences d'un caractère dans une chaîne :

```

1 int compte(QString texte, QChar c)
2 {
3     if (texte.isEmpty())
4         return 0;
5     int onEnTientUn = (texte.left(1) == c) ? 1 : 0;
6     return onEnTientUn + compte(texte.right(texte.length()-1),c); //récursion
7 }
```

Etant donné que la fonction `QString::contains()` est parfaitement capable de faire ce travail, la fonction `compte()` est évidemment sans intérêt pratique...

⁴ La communication descendante est néanmoins assurée par un passage de paramètre, en l'occurrence le pointeur `this` qui, dans l'exécution enchâssée, aura pour valeur l'adresse de l'instanceLocale.

La fonction `compte()` commence par vérifier (3) si la chaîne est vide. Si c'est le cas, le nombre d'occurrences du caractère `c` y est nécessairement nul, et la fonction renvoie donc zéro (4). Dans le cas contraire, la fonction calcule le "nombre d'occurrences du caractère `c`" dans le premier caractère de la chaîne. La réponse recherchée est alors simplement la somme de ce nombre et du nombre d'occurrences du caractère `c` dans le reste de la chaîne. La variable locale `onEnTientUn` permet donc à chacune des exécutions enchâssées de stocker le résultat du traitement d'un caractère, la somme de ces résultats étant effectuée lors de la remontée récursive. Remarquez qu'aucune variable ne contient jamais cette somme, car il ne s'agit que du résultat de l'évaluation d'une expression, résultat que le mécanisme de renvoi propage jusqu'à la fonction appelante. A titre d'illustration, l'appel

```
cout << compte("Il était une fois un petit chaperon rouge", 't');
```

produirait évidemment l'affichage du nombre 4.

5 - Un appel récursif peut simplifier l'écriture d'un programme

Il y a fort à parier que, pour la plupart des lecteurs de cette Leçon, la façon naturelle d'écrire la fonction `compte()` est actuellement :

```
1 int compte(QString texte, QChar c)
2 { //renvoie le nombre d'occurrences de c dans la chaîne p
3   int compteur = 0;
4   int position;
5   for(position = 0 ; position < texte.length() ; ++position)
6     if(texte[position] == c)
7       compteur = compteur + 1;
8   return compteur;
9 }
```

Ce qui nous laisse face à une question inévitable : à quoi bon écrire des fonctions récursives ?

S'il ne s'agit que de rendre le code difficile à comprendre pour le lecteur, le langage C++ offre d'autres moyens, tout aussi efficaces qu'un appel récursif :

```
int compte(QString texte, QChar c)
{ //renvoie le nombre d'occurrences de c dans la chaîne p
  const char *p = texte.latin1();
  int n;
  for(n = 0 ; *p ; n += *p++ == c.latin1());
  return n;
}
```

La meilleure réponse est à mon avis la suivante : si vous voyez comment écrire une boucle qui traite le problème, inutile de vous forcer à imaginer une solution récursive.

Cette règle possède un corollaire selon lequel, si vous voyez comment écrire une fonction récursive qui traite le problème, il n'est peut être pas indispensable de vous forcer à imaginer une solution purement itérative.

Avant d'affirmer que ce corollaire ne vous concernera jamais, voyez le TD 15 et la Leçon 20...

6 - Bon, c'est gentil tout ça, mais ça fait quand même 7 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) Une instruction qui appelle la fonction dans laquelle elle figure est un appel récursif.
- 2) Une fonction contenant un appel récursif incondtionnel crée une récursion infinie.
- 3) Comme le jeu d'échecs, l'usage d'appels récursifs est capable, à partir d'un petit nombre de règles simples, de créer des situations relativement complexes.
- 4) Une grande partie du secret d'une fonction récursive réside dans la nature exacte de ses paramètres : tous les niveaux d'exécution travaillent-ils sur les mêmes objets ?

- 5) Lorsque la fonction récursive est membre d'une classe, la réflexion suggérée au point précédent doit prendre en compte le paramètre caché `this`.
- 6) Le reste du secret d'une fonction récursive réside vraisemblablement dans la valeur qu'elle renvoie.
- 7) Contrairement à ce que les remarques précédentes pourraient laisser croire, il existe des cas où utiliser un appel récursif est la solution la plus simple.