# Série 6: Fonction (2)

## Récursivité, surcharge, valeur par défaut, coût calcul

## Lien avec le MOOC Initiation à la Programmation (en C++)
## Lien avec ICC-Théorie en complément du MOOC

Les éléments du MOOC sur les fonctions sont répartis sur deux semaines. Cette seconde partie se concentre sur la conception de fonctions récursives.

Le premier exercice complémentaire porte sur la mesure du coût calcul effectif pour une exécution donnée d'un exécutable. Le second sert à illustrer le complément orienté projet N°2.

Le complément orienté projet N°2 porte sur la redirection de l'entrée et de la sortie par défaut.

## Exercices partiels semaine4 du MOOC

- **Document Tutoriel 2ième partie seulement « Somme récursive »**
  - Écriture d'une fonction récursive : ne pas oublier le critère d'arrêt

- **Document Exercices semaine 4 du MOOC : 2ième sélection**

  - **Exercice 14 : nombres de Fibonacci (niveau 1)**
    - Permet de comparer les performances de l'approche récursive avec une solution itérative. Cet exercice est ré-utilisé pour l'exercice complémentaire ExC 1 sur la mesure du temps de calcul en p 2.

- **Document Exercices additionnels semaine 4 du MOOC : 2ième sélection**

  - **Exercice 11 : Recherche dichotomique (niveau 2)**
    - Essayez les deux variantes suivantes de structure de contrôle pour le test du caractère alphanumérique. L'approche avec une cascade de `if- else if -`… peut être remplacée par un **switch** sur le caractère lu au clavier car le type **char** est automatiquement converti en **int** dans l'évaluation d'une expression en C++.

# Exercices Complémentaires (ExC)

**ExC 1 : Mesure du temps de calcul d'un morceau de code sans entrée-sortie (niveau 2)**

Nous avons vu plusieurs solutions possibles pour un certain nombre d'exercices. Parfois la différence de performance est clairement exprimée par un ordre de complexité différent qui se manifeste par une durée d'exécution perceptible à l'échelle humaine. Dans d'autres cas, l'ordre de complexité peut être le même et nous ne pouvons pas juger les performances d'exécution du code sur des temps court à notre échelle.

C'est pourquoi nous présentons ici une méthode simple pour obtenir une estimation relativement fiable du temps d'exécution d'un morceau de code. Ce morceau de code ne doit pas contenir d'entrée-sortie car cela fausse la mesure. Pour cela nous devons inclure ce fichier en-tête :

```
#include <ctime>
```

Le fichier en-tête **ctime** offre la fonction **`clock()`** qui renvoie une valeur entière signée indiquant le temps CPU exprimé en unité d'horloge avec le type **`clock_t`** depuis le début de l'exécution du programme. En cas d'erreur la fonction **`clock()`** renvoie **-1** ; il faut tester cette valeur et réagir en conséquence pour éviter de produire de fausses estimations. On travaille toujours sur la différence entre deux appels de cette fonction, un avant et un après la portion de code à tester.

La durée correspondant à l'unité exprimée par le type **`clock_t`** n'est pas standardisée. Pour convertir une durée de **`clock_t`** en seconde il faut la diviser par le symbole **`CLOCKS_PER_SEC`** qui est défini dans **`<ctime>`**. Sur les VM, une unité **clock_t** correspond à ***1 microseconde***, ce qui est bien plus long que le temps d'exécution du morceau de code qui nous intéresse. Donc on doit répéter le code dans une boucle qui l'exécute **NFOIS** de manière à avoir une durée mesurée > 0 ; ensuite on divise par **NFOIS** pour obtenir la véritable durée du code testé. L'évaluation doit être répétée plusieurs fois pour disposer de chiffres fiables.

```
#include <ctime>
constexpr int NFOIS(1000) ; // à ajuster selon le code évalué

int start(0);
do {
    start = clock() ;
} while(start == -1) ;

for(int k(0) ; k < NFOIS ; ++k)
{
     ici le code SANS ENTREE-SORTIE
     dont on veut mesurer la durée d'exécution
}
int end(clock()) ;
if(end == -1){
      cout << "invalid clock tick" << endl ;
      exit(0);
}
int nb_cycles = end - start ;
```

Il faut diviser **nb_cycles** par **NFOIS** et par le symbole **CLOCKS_PER_SEC** pour obtenir une durée en secondes. Cette valeur étant très petite pour la plupart des valeurs de n, il vaut mieux l'exprimer en **nanoseconde** en multipliant par **1e9**.

```
double duree_en_ns = (1e9 / CLOCKS_PER_SEC )*nb_cycles / NFOIS ;
```

Travail à effectuer : comparer le temps calcul mesuré avec cette approche sur le code du calcul des coefficients de Fibonacci de l'exercice 14 du MOOC (mais SANS affichage), successivement avec l'approche itérative puis récursive (pas les deux en même temps).

1) Effectuez les mesures pour les valeurs de n variant de 2 à 20 par pas de 2.

2) Avez-vous les même mesures que des personnes travaillant sur leur laptop (ou vice-versa) ?

3) Comment la performance évolue-t-elle pour les cas particulier du doublement de la taille du problème (par exemple entre n valant 10 et 20) ? Est-ce cohérent avec l'ordre de complexité de ces deux implémentations ?

**ExC 2 : Codage de César / manipulation du code des caractères alphanumériques**

Jules César codait ses messages secrets en décalant les lettres dans l'alphabet d'un nombre fixe de lettres. Nous allons faire une version de ce codage en décalant le code ASCII des caractères d'un texte de **n** caractères dans l'alphabet. Le même programme peut servir pour coder et décoder un message. Pour le décodage il suffit de recoder le message déjà codé mais cette fois en indiquant un décalage de **-n**.

On va se restreindre à des messages contenant des lettres MAJUSCULES, des espaces, le point séparant des phrases et le passage à la ligne. il ne faudra pas coder l'espace ' ', le point '.' et le passage à la ligne '\n'.

Travail à effectuer : le programme doit d'abord lire la valeur du décalage = un entier entre -13 et + 13.

Ensuite le programme lit le texte fourni en entrée caractère par caractère avec **cin** et affiche aussitôt avec **cout** chaque caractère codé avec le décalage (ou pas). La détection de tout autre caractère que ceux qui sont prévus agit comme un signal de fin de message et termine le programme.

Pour le codage, le programme doit veiller à reboucler le code à l'autre bout de l'alphabet en cas de dépassement du début ou de la fin de l'alphabet. Il est demandé de ne pas utiliser les valeurs numériques brutes du code ASCII dans les tests effectués. Travaillez seulement avec les constantes littérales de type char, par exemple **'Z'**.

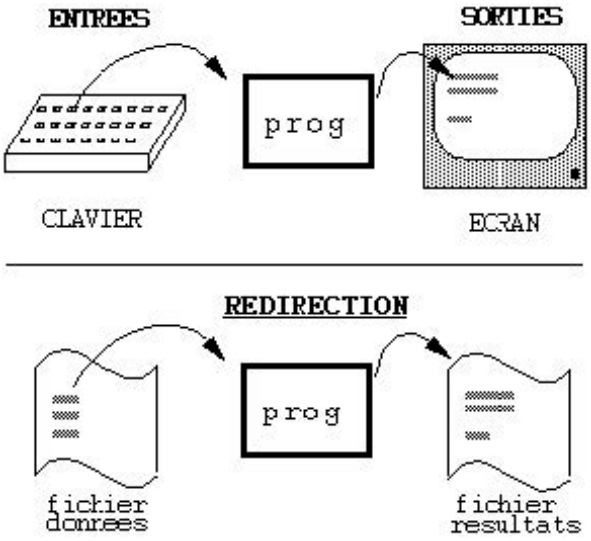
Exemple d'exécution : on donne d'abord 1 comme valeur de décalage puis on entre le texte HAL suivi par Enter ce qui va lancer le codage de la première ligne et afficher IBM. Ensuite on peut quitter le programme en tapant une lettre minuscule puis Enter.

Mise en œuvre du principe d'abstraction : mettez en œuvre une fonction responsable du codage en plus de main. Elle renvoie VRAI si le caractère passé par référence a pu être codé et FAUX s'il faut terminer le programme.

Question subsidiaire : trouver un codage intéressant de GOOGLE

# Complément en prévision du Projet (2)

**Redirection de l'entrée (clavier) et de la sortie (terminal) par défaut.**




Dans les exercices précédents, les données en entrée du programme étaient toujours lues au clavier, et les sorties toujours écrites à l'écran. C'est bien mais on peut faire beaucoup mieux sans changer une seule ligne de notre code source…

En particulier la phase de mise au point d'un projet demande de faire beaucoup de tests différents pour s'assurer que le programme fait ce qu'il doit faire. Entrer manuellement toutes ces données au clavier prend énormément de temps.

La ***redirection de l'entrée*** permet de prendre le contenu d'un fichier texte comme si cela venait du clavier. La méthode de travail est donc d'écrire une fois pour toutes vos fichiers de tests avec geany puis de rediriger le fichier de test qui vous intéresse sur l'entrée par défaut de votre programme exécutable. Supposons qu'il s'appelle **prog** et que les données à rediriger vers l'entrée sont dans le fichier **donnees.txt** ; il suffit de taper la commande suivante dans une fenêtre Terminal :

**./prog < donnees.txt**

Inversement, la sortie affichée dans le terminal est parfois très longue ; il est plus pratique de la rediriger vers un fichier puis d'ouvrir ce fichier avec geany pour l'examiner en détail. Si vous voulez ***rediriger la sortie*** du programme **prog** vers un fichier **resultats.txt**, il suffit de taper :

**./prog > resultats.txt**

Vous pouvez aussi combiner ces deux redirections: ./**prog < donnees.txt > resultats.txt**

**Remarque**: en utilisant le symbole >> au lieu de >, vous ajoutez les résultats à la fin d'un fichier déjà existant.

Travail à effectuer : écrivez un petit fichier texte de test avec geany destiné à votre programme de codage de César (ExC 2). Ce fichier doit commencer par l'entier indiquant le décalage, puis respecter les règles indiquées pour le message et enfin finir avec un caractère qui produit la fin du programme.

Ensuite, depuis une fenêtre terminal, redirigez ce fichier texte vers le programme du codage secret et redirigez la sortie vers un autre fichier texte.

Éditez-le pour indiquer l'opposé du décalage précédent en début de fichier. Recommencez le codage et vérifiez si vous obtenez bien le message original.