



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

Apprendre C++ avec Qt : Annexe 6

Fonction inline

Lorsqu'une fonction dépourvue d'argument est appelée, la séquence de code exécutée ressemble beaucoup à celle qui aurait été obtenue en insérant le code définissant cette fonction à la place de l'instruction qui l'a appelée. Imaginons par exemple que deux fonctions soient définies de la façon suivante :

```
1 void fonctionAppelee()  
2 {  
3     cout << " Je suis là !";  
4 }
```

```
1 void fonctionAppelante()  
2 {  
3     cout << "Début de l'exécution " << endl;  
4     fonctionAppelee();  
5     cout << "Fin de l'exécution " << endl;  
6 }
```

Le résultat obtenu lors de l'exécution ressemble fort à celui qu'aurait produit l'exécution de

```
1 void fonctionUnique()  
2 {  
3     cout << "Début de l'exécution " << endl;  
4     cout << " Je suis là !";  
5     cout << "Fin de l'exécution " << endl;  
6 }
```

Du point de vue de l'écriture du programme, les deux situations sont très différentes, surtout si la fonctionAppelee() est un peu complexe et est appelée plusieurs fois. Par ailleurs, si la fonctionAppelee() possède des paramètres, le code qui pourrait remplacer son appel dépend des valeurs transmises.

Du point de vue de l'exécution, en revanche, la différence essentielle est que la première approche impose une sauvegarde de l'état d'avancement de l'exécution de la fonctionAppelante(), l'initialisation des éventuels paramètres de la fonctionAppelee() avec les valeurs transmises, puis une restauration de l'état d'avancement de la fonctionAppelante() une fois l'exécution de la fonctionAppelee() terminée.

Lorsque la fonction est très brève, ces opérations "administratives" ont un coût qui cesse d'être négligeable par rapport à la durée d'exécution de la fonction. Pour éviter aux programmeurs la tentation d'essayer d'obtenir une exécution plus rapide en s'abstenant de créer ce genre de fonctions, le langage C++ permet de demander que certaines fonctions ne soient pas réellement appelées, mais que le compilateur remplace effectivement leur appel par le code équivalent. Il suffit pour cela que, lors de sa définition, le nom de la fonction soit précédé du mot inline :

```
1 inline void fonctionAppelee()  
2 {  
3     cout << " Je suis là !";  
4 }
```

Il faut toutefois souligner qu'il ne s'agit là que d'un souhait exprimé par le programmeur. Le compilateur reste, en tout état de cause, l'arbitre de ce qui peut ou non être réalisé "inline".

La notion de fonction inline appelle en outre un dernier commentaire : les fragments de code ainsi définis n'ont plus de fonction que le nom, puisque le mécanisme que leur invocation déclenche n'est précisément pas celui d'un appel de fonction. Cette différence a une conséquence pratique importante : le compilateur doit disposer du code correspondant à la

"fonction" au moment où il traite un "appel" à celle-ci, puisque cet appel doit justement être remplacé par le code en question. Le principe d'autonomie de compilation des `.cpp` interdit donc que les fonctions `inline` soient définies dans des fichiers `.cpp`. Par conséquent,

Lorsqu'une fonction est `inline`, sa définition doit figurer dans un `.h` et non dans un `.cpp`

Bien entendu, ce fichier `.h` devra être inclus dans tout `.cpp` où figure un appel à la fonction `inline` concernée. Comme une fonction `inline` n'est pas véritablement une fonction, la présence de sa définition dans un `.h` ne pose pas de problème de redéfinition en cas d'inclusions multiples de ce `.h` dans un projet. C'est d'ailleurs cette tolérance à la redéfinition présentée par les fonctions `inline` qui explique qu'on puisse définir une fonction membre au cours de la définition d'une classe (c'est à dire dans un `.h`, qui a vocation à être inclus plusieurs fois dans le même projet). En effet,

Lorsqu'une fonction membre est définie à l'intérieur la définition de la classe à laquelle elle appartient, sa définition est automatiquement considérée comme étant `inline`.