TD-3
Recursivite

Recursivite Simple

Lors du Td precedent nous avons vu que *De facon simpliste, une procédure (ou une fonction) est un moyen de factoriser un code souvent utilisé et d'utiliser ce bloc simplement à un autre endroit du programme en l'appelant par son nom. En réalité c'est un mécanisme plus intressant que le simple recopiage car on peut utiliser le nom de la procédure dans le code même de la procédure que l'on est en train de définir.*

Puis nous nous etions interesse essentiellement aux variables et parametres. Cette distinction entre fonctions et factorisation est illustree ici :

Exercice 1. :
Que fait le programme suivant sur des entrees positives ? sur des entrees negatives ? Corriger le pour qu'il ait un comportement coherent pour toute entrees. Dessiner l'arbre des appels pour ce programme. Comment progresse le point de controle dans cet arbre ? A un moment donne quelles sont les procedures en cours ?

```
#include <iostream>

void f (int n){
  cout << n << ‘‘ ‘‘;
  if (n!=0) f (n-1);
  cout << n << ‘‘ ‘‘;
}

int main(){
  int n;
  cin >> n;
  f(n);
  cout << endl;
}
```

Exercice 2. :
Que fait la fonction suivante :

```
int f(int n){
 if (n<0) return f(-n);
 if (n==0) return 0;
 return (1+f(n/10));
}
```

Exercice 3. :

1. Ecrire une fonction qui calcule la factorielle d'un entier.
2. Ecrire une fonction qui teste si un entier est divisible par un autre.
3. Ecrire une procedure qui affiche la decomposition d'un entier en produit de facteur premier.

4. On rappelle qu'un entier est divisible par 3 si la somme de ses chiffres est divisible par 3. Ecrire une fonction recursive, en identifiant bien les cas de base, qui utilise cette propriete pour dire si un nombre est divisible par 3.

5. Connaissez vous une regle analogue pour la divisibilite par 11 ? Bon, alors la voila : si on numerote les chiffres de l'ecriture d'un nombre pour leur attribuer une position, disons de droite a gauche, alors si la somme des chiffres en position paire est egal a la somme des chiffres en position impaire c'est que le nombre est divisible par 11. Ecrire une fonction qui utilise cet argument pour tester la divisibilite par 11.

Recursivite multiple

Exercice 4. Que calculent les fonctions suivantes :

```
#include <iostream>

// prototypes
   bool odd (int i);
   bool even(int i);

// corps des fonctions
   bool even (int i) {
    if (i==0) {return true;}
    else {return (odd(i-1));}
   }

   bool odd (int i) {
    if (i==0) {return false;}
    else {return (odd(i-1));}
   }

   void main(){
   cout << odd(5) << odd(^) << even (5) << even(6) << odd(-007) << endl;
   }
```

Pour ecrire des fonctions mutuellement recursive, on a indiqué au compilateur les profils, ou prototypes, des fonctions avant de les ecrire. C'est pour lui une aide a l'analyse statique.

La declaration des prototypes n'est pas une obligation theorique, on peut trouver des langages ou elle n'est pas necessaire. Par contre elle l'est pour le c++ que l'on utilise. Vous pouvez comparer cette obligation a l'indicage des tableaux en c++ : ce sont obligatoirement des entiers qui commencent a zero, alors qu'on pourrait imaginer des tableaux dont les indices sont un intervalle [-5,3] ou des enumerations. Gardez a l'esprit que $c++$ est un compromis.

Graphe de dependence

On dit qu'une fonction f appelle une fonction g si la definition de f comporte un appel a la fonction g. Le *graphe de dependence* d'un ensemble de fonction V est le graphe oriente dont l'ensemble des noeuds est V, et où un arc relie un noeud $f \in V$ à un noeud g si f appelle g.

Exercice 5. Donner une definition de la recursivite basée sur le graphe de dépendence, dessiner le graphe de dependance pour les fonctions $main, odd, even$. Quel est le rapport avec l'arbre des executions ?

Exercice 6. La procedure suivante est elle recursive :

```
void  f(){
 if (2 > 3) f();
}
```

Exercice 7. Que pensez vous du programme suivant ?

```
#include <iostream>
void main(){
  int i;
  cin >> i;
  if (i>0) main();
}
```

Exercice 8. Reecrire la fonction *even* de sorte qu'elle ne contienne que des appels recursifs a *even*. Une generalisation est possible : si un programme contient un ensemble de fonctions V, il existe un programme equivalent qui ne contient que *main* et une autre fonction recursive. Avez vous une idee de comment ?

Exo 1: Si l'entier entré est positif, par exemple 5, le programme affichera "5 4 3 2 1 0 0 1 2 3 4 5". Il ne s'arretera pas sur une entree negative. On peut corriger de plusieurs facon,

```
void f (int n){
  cout << n << ‘‘ ‘‘;
  if (n > 0) f (n-1);
  cout << n << ‘‘ ‘‘;
}
```

ou encore :

```
void f (int n){
  cout << n << ‘‘ ‘‘;
  if (n > 0) f (n-1);
  if (n < 0) f (n+1);
  cout << n << ‘‘ ‘‘;
}
```

Exo 2: Il calcule la taille de la representation decimale de n. Demander d'ecrire la meme en base 9.

Exo 3 : Pour la factorielle :

```
#include <iostream>

int fact(int n){
if (n<=0) return 1;  // On prend comme convention que la factorielle de 0 et d'un negatif c'est 1
return n*fact(n-1);
}

int main(){
 int i;
 cout << ‘‘entrez une valeur’’;
 cin >> i;
 cout << ‘‘la factorielle de ’’ << i << ‘‘ est ’’ << fact(i) << endl;
}
```

Pas besoin de recursivite pour la divisibilite.

```
bool divise(int a, int b){   // est ce que b divise a ?
 return ((a mod b)==0);
}
}
Quoi que on peut ecrire :
{\small \begin{verbatim}
bool divise (int a, int b){ // est ce que b divise a ?
 if (a<0) return divise(-a,b);
 if (b<0) return divise(a,-b);
 if (a==0) return true;
 if ( (0< a) && (a<b) ) return false;
 return (divise(a-b,b));
}

void decompose (int n) {
 if (n <  0) {
   decompose (-n);
   return;
 }
 if ((n == 0)||(n==1)) {
  cout << n << endl;
  return;
 }
 int i=2;
 while( !divise(n,i)) i++;
 cout << i << ‘‘ ‘‘;
 decompose(n/i);
}
```

Une fonction recursive pour la somme des chiffres :

```
int somme(int i){
 if (i<0) return somme(-i);
 if ( (0<= i) && ( i<=9)) return i;
 return (i mod 10) + somme (i/10);
}
```

Une autre pour la divisibilite par 3 :

```
bool divisible_par_3(int i){
 int s;
 if (i<0) return divisible_par_3(-i);
 s=somme(i);
 if ( (s==0) || (s==3) || (s==6) || (s==9) ) return true;
 if ( s<10) return false;
 return (divisible_par_3(s));
}


bool divisible_par_11(int i){
 int sp=0;
 int si=0;
 calcule_sommes(i,sp,si);
 return (sp==si);
}

void calcule_sommes(int i,int &si,int &sp){
 if (i==0) return;
 sp=sp+ (i mod 10);  // je numerote les chiffres en partant de zero a droite
 si=si+ (i/10 mod 10);
 calcule_sommes(i/100,si,sp);
}
```

Exo 4 : Une fonction est dite recursive si elle appartient a un cycle du graphe de dependence.

Exo 5: Oui, il faut comprendre qu'on a choisit de definir la recursivite comme une notion statique. Je n'en rajoute pas, mais ca entraine une discussion on pourrait se poser la meme question pour une fonction dont on sait par avance que la profondeur des appels est bornee par une constante. On voit alors que pour exprimer une recursivite "pure" il faudrait savoir resoudre des problemes tres compliques. Ce qui peut expliquer que la definition de recursivite soit statique, pour ne pas mettre le doigt dans qq chose qui mene trop loin.