Feuille IV

Exercice 1: Donnez **l'arbre des portées** du programme ci-dessous. On exécute le programme : donnez l'état de la mémoire et de l'environnement, selon le formalisme du cours, juste après l'exécution de chacune des instructions marquées `(†)` . Indiquez ce qu'imprime le programme.

```
int x;
int a;
void act1(int a) {
  x = x + a;
  a = a + 2;                (†)
}

void act2(int a, int b) {
int x;
  x = a + 3 * b;            (†)
  act1(x);
  a = 2 * x + b;            (†)
}

int main() {
  x = 1; a = 2;              (†)
  act2(x, x);
  x = x + a;                 (†)
  cout << "valeur: " <<  x << ", a" << a << endl;
  return 0;
}
```

Exercice 2: Écrire une version récursive **et** une version non récursive d'une fonction qui imprime la représentation binaire d'un entier positif ou nul (de type `int`). Le bit de poids faible doit bien sûr être à droite ! Exemples :

```
Representation binaire de 0:  0
Representation binaire de 63: 111111
Representation binaire de 64: 1000000
Representation binaire de 65: 1000001
```

Exercice 3 : On définit les *nombres de Stirling de seconde espèce* S(n, k), 0 ≤ n, k par les relations de récurrence ci-dessous. Le tableau donne les premières valeurs (S(n, k) vaut 0 pour k > n).

S(n, k) = S(n-1, k-1) + k*S(n-1, k) pour n, k > 0
S(0, 0) = 1
S(n, 0) = S(0, n) = 0, pour n > 0

n \ k	**0**	**1**	**2**	**3**	**4**	**5**
0	1					
1	0	1				
2	0	1	1			
3	0	1	3	1		
4	0	1	7	6	1	

1. Écrire une fonction **récursive** de calcul de S(n, k) : `int stirling(int n, int k);`
2. Pourquoi n'est-il pas efficace de calculer les nombres de Stirling de cette façon ?

3. (Plus dur) Donner une version itérative du calcul, en calculant à chaque fois une ligne à partir de la ligne précédente. Le calcul doit se faire de la droite vers la gauche si on veut pouvoir n'utiliser qu'un seul vecteur puisque S(n, k) référence S(n-1, k-1).

Exercice 4: On considère le programme suivant (l'opérateur `/` représente la division entière et `%` le modulo; par exemple `2014 / 100` vaut `20` et `2014 % 100` vaut `14`).

```
int cste = 60;

void decompose(int val, int &div, int &mod) {
   div = val / cste;
   mod = val % cste;          // instruction référencée  †
}

int main() {
   int v, m, h, d;
   decompose(6004, v, m);
   if (v != 0) {        // Bloc interne B
     int cste = 24;
     decompose(v, d, h);
   }
   cout << "d: " << d << "h: " << h << "m: " << m << endl;
   return 0;
}
```

Donner l'arbre des portées du programme. Indiquer l'état de la mémoire et de l'environnement juste après l'exécution de l'instruction notée † à chacun des deux appels à `decompose`. Qu'affiche le programme ? Pourquoi la fonction `decompose` utilise-t-elle des références ?

Exercice 5 : On considère les fonctions ci-dessous :

```
int & f(int &px, int &py, int &pz) {
  if (px < py) {
    if (px <= pz) { return px;} else { return pz; }
  } else {
    if (py <= pz) { return py; } else { return pz; }
  }
}

int & g() {
  int x = 3, y = 9, z = 7;
  int &p = f(x, y, z);
  p = 2*p;
  return p;
}

int main() {
  int &ref = g();
  cout << "Valeur: " << ref << endl;
  return 0;
}
```

A. Indiquer l'environnement et l'état mémoire qui existent au moment d'exécuter l'instruction `return` dans chacune des fonctions `f` et `g`.

B. Cet ensemble de fonctions pose-t-il un problème de **durée de vie** des variables ? Si « oui », expliquer où et pourquoi, si « non » expliquer pourquoi.

C. Pourquoi la fonction `f` renvoie-t-elle une référence sur un entier plutôt qu'un entier ?

Exercice 6 : On appelle *distance minimale d'édition* entre deux chaînes de caractères *x* et *y* le nombre minimal de transformations élémentaires (insertion, suppression ou remplacement d'un caractère par un autre) permettant de transformer *x* en *y*.

Exemples :

`dist("ababb", "babaaa") = 3`

Ajouter un `b` *initial et remplacer les deux* `b` *de queue par des* `a`*.*

`dist("retiree", "lettre") = 3`

Remplacer le `r` *initial par* `l`*, remplacer* `i` *par* `t`*, supprimer le* `e` *final.*

Il peut y avoir plusieurs séquences différentes de même taille.

Écrire une fonction **récursive** pour calculer la distance minimale d'édition entre deux chaînes de caractères. À chaque étape on regarde les nombres des différentes transformations possibles et on renvoie la meilleure.

Remarque : la version récursive est coûteuse; il existe une version non récursive beaucoup plus efficace en utilisant une technique dite de « programmation dynamique ».

(Optionnel, plus dur) : en plus de calculer la distance, renvoyer la séquence de commandes pour passer d'une chaîne à l'autre pour cette distance minimale d'édition.

Exercice 7: Que calcule la fonction f ci-dessous, définie pour les entiers positifs ou nuls ? (on examinera les valeurs de x de façon décroissante à partir de 100)

```
int f(int x) {
  if (x > 100) {
    return x - 10;
  } else return f(f(x+11));
}
```