TESTC++final Sem1 MT & EL 17.12.2021 EPFL / R. Boulic

W=blanc, B=bleu, S=saumon, Y=jaune, All = même réponse pour tous

1) (3 pts) : Opérateurs bit-à-bit

Ecrire une fonction qui renvoie **true** seulement si les deux paramètres de type **char** *sont différents sur un seul bit*. Peu importe le bit qui est différent. La fonction doit renvoyer **false** si *aucun bit n'est différent* ou si *plus de un bit sont différents*.

On demande d'utiliser les opérateurs bit-à-bit pour accéder individuellement aux 8 bits des paramètres et évaluer ce booléen.

Voici quelques exemples de valeurs binaires des paramètres et du résultat attendu :

- Pour **01100101** et **00100101** la fonction renvoie **true** car *un seul bit est différent* entre ces 2 motifs binaires (c'est le second bit du coté des poids forts)

- Pour **01100101** et **00100111** la fonction renvoie **false** car *plus de un bit sont différents* entre ces 2 motifs binaires (c'est le second bit du coté des poids forts et le second bit du coté des poids faibles)

- Pour **01100101** et **01100101** la fonction renvoie **false** car *aucun bit n'est différent* entre ces 2 motifs binaires.

Comme il est préférable de travailler avec des opérandes de type **unsigned** pour les opérateurs bit-à-bit, nous avons déclaré deux variables **a** et **b** de ce type et nous les avons initialisées avec le motif binaire des deux paramètres. La fonction doit seulement examiner les 8 bits de poids faibles de **a** et **b** pour compléter sa tâche. Vous pouvez déclarer vos propres variables locales.

Compléter le code de la fonction (environ 6 lignes suffisent) **[All]**

```
1   bool single_bit_difference(char octet_a, char octet_b)
2   {
3         unsigned a(octet_a), b(octet_b);
4         unsigned difference_count(0);
5
6         // pour extraire un seul bit
7         unsigned mask(1);
8
9         for ( int i(0); i < 8; i++ )
10        {
11              if ( (a & mask) != (b & mask) )
12                    difference_count++;
13
14              // décalage d'un bit vers la gauche
15              mask = mask << 1;
16        }
17        return difference_count == 1;
18
19  }
```

2) (5 pts) vector de string

Ce programme compile en C++11 sans warning

```
1   #include <iostream>
2   #include <vector>
3   #include <string>
4   using namespace std;
5
6   vector<string> f1(vector<string> string_dict, int index1, int index2);
7   void           f2(vector<string> string_dict, int index1, int index2);
8   string*        f3(vector<string> string_dict, int index1, int index2);
9
10  vector<string> f1(vector<string> string_dict, int index1, int index2)
11  {
12    string temp;
13    temp = string_dict[index1];
14    string_dict[index1] = string_dict[index2];
15    string_dict[index2] = temp;
16    return string_dict;
17  }
18
19  void f2(vector<string> string_dict, int index1, int index2)
20  {
21    string temp;
22    temp = string_dict[index1];
23    string_dict[index1] = string_dict[index2];
24    string_dict[index2] = temp;
25  }
26
27  string *f3(vector<string> string_dict, int index1, int index2)
28  {
29    string *p;
30    p = &string_dict[index1] + (index2-index1);
31    return p;
32  }
33
34  int main()
35  {
36    vector<string> string_dict;
37    string_dict.push_back("123");
38    string_dict.push_back("456");
39    string_dict.push_back("789");
40    int a(0), b(1), c(2);
41
42    string_dict = f1(string_dict, a, b);
43    // Question 2.1
44
45    string_dict = f1(string_dict, b, c);
46    // Question 2.2
47
48    f2(string_dict, c, a);
49    // Question 2.3
50
51    string *p(nullptr);
52    p = f3(string_dict, a, b);
53    // Question 2.4
54
55    p = f3(string_dict, a, c);
56    *p = string_dict[1];
57    // Question 2.5
58
59    return 0;
60  }
```

W=blanc, B=bleu, S=saumon, Y=jaune, All = même réponse pour tous

Page suivante, on demande de compléter la table des valeurs du vector **string_dict** pour certaines lignes de code indiquées ci-dessus. La première ligne de la table est déjà remplie.

[W] la ligne 41 correspond aux initialisations des lignes 37 à 39 de la page précédente

Valeur de :	`string_dict[0]`	`string_dict[1]`	`string_dict[2]`
Ligne 41	123	456	789
Ligne 43	456	123	789
Ligne 46	456	789	123
Ligne 49	456	789	123
Ligne 53	456	789	123
Ligne 57	456	789	789

[B] la ligne 41 correspond aux initialisations des lignes 37 à 39 de cette variante

Valeur de :	`string_dict[0]`	`string_dict[1]`	`string_dict[2]`
Ligne 41	987	654	321
Ligne 43	654	987	321
Ligne 46	654	321	987
Ligne 49	654	321	987
Ligne 53	654	321	987
Ligne 57	654	321	321

[Y] la ligne 41 correspond aux initialisations des lignes 37 à 39 de cette variante

Valeur de :	`string_dict[0]`	`string_dict[1]`	`string_dict[2]`
Ligne 41	789	456	123
Ligne 43	456	789	123
Ligne 46	456	123	789
Ligne 49	456	123	789
Ligne 53	456	123	789
Ligne 57	456	123	123

[S] la ligne 41 correspond aux initialisations des lignes 37 à 39 de cette variante

Valeur de :	`string_dict[0]`	`string_dict[1]`	`string_dict[2]`
Ligne 41	321	654	987
Ligne 43	654	321	987
Ligne 46	654	987	321
Ligne 49	654	987	321
Ligne 53	654	987	321
Ligne 57	654	987	987

3) (5 pts) tableau de chaîne à-la-C

Ce programme compile en C++11 sans warning

```
1   #include <iostream>
2   #include <cstdlib>
3   #include <string>
4   using namespace std;
5
6   int f(const char *a)
7   {
8      int val(0), digit_val(0), i(0);
9      while(a[i] != '\0')
10     {
11        digit_val = a[i] - '0';
12        if(digit_val < 0 or digit_val > 9) exit(0);
13        val = 10*val + digit_val;
14        i++ ;
15     }
16     return val;
17  }
18
19  int main(int argc, char *argv[])
20  {
21     string message("the sum is ");
22     int value(0);
23
24     if(argc == 1)
25     {
26        cout << "I need more" << endl;
27        return 0;
28     }
29     else
30     {
31        int n(f(argv[1]));
32        value = n * (n + 1) / 2;
33        if(argc == 3)
34        {
35           string operation(argv[2]);
36           if(operation == "prod")
37           {
38              message = "the product is ";
39              value = 1;
40              for(int i(1); i <= n; i++) value *= i;
41           }
42        }
43     }
44     cout << message << value  << endl;
45     return 0 ;
46  }
```

En supposant que l'exécutable s'appelle **prog**, on demande ce qu'affiche le programme quand il est exécuté dans le terminal avec les lignes de commande suivantes suivies de Enter :
(on demande de justifier chaque affichage obtenu ; la justification du but de la fonction **f()** n'a besoin d'être faite que la première fois où elle est utilisée) :

3.1) **`./prog`** **`[All]`**

Affiche **`I need more`** car **argc** vaut 1 puisque aucun argument n'est transmis en plus du nom de l'exécutable

3.2) **`./prog 10`** **`[All]`**

Un seul argument est transmis après le nom de l'exécutable, donc **argc** vaut **2**.
La chaîne à-la-C **argv[1]** est passée en argument à la fonction **f().**
Cette chaîne contient le code ASCII des 3 caractères suivants : `1 puis 0 puis \0` respectivement pour les indices 0, 1 et 2. Seuls les 2 premiers caractères produisent une condition vraie pour entrer dans la boucle while :
Premier passage :
les codes ASCII étants consécutifs **'1' – '0'** donne la valeur **1** pour **digit_val**.
La variable **val** initialisée à 0 en dehors de la boucle prend la valeur **1**.
Le compteur **i** est incrémenté à **1**
Second passage :
'0' – '0' donne la valeur **0** pour **digit_val**.
La variable **val** prend la valeur **10**.
Le compteur **i** est incrémenté à **2**

La fonction f() renvoie la valeur entière **10**
Ligne 32 : value prend la valeur 10*11/2 = **55**
Comme argc vaut 2 on passe directement à la ligne 44 qui affiche : **`the sum is 55`**

3.3) **`./prog 4 prod`** **`[W et S]`**

f () renvoie l'entier **4**
Cette fois-ci **argc** valant **3** une comparaison est effectuée entre la string initialisée avec la chaîne à-la-C argv[2] et la constante **`prod`**. La condition étant vraie, la string **message** est modifiée avec **`the product is`** et **value** est modifiée avec la valeur **1*2*3*4** valant **24**.
Ligne 44 : on affiche : **`the product is 24`**

3.3) **`./prog 5 prod`** **`[B]`**

f() renvoie l'entier **5**
Cette fois-ci **argc** valant **3** une comparaison est effectuée entre la string initialisée avec la chaîne à-la-C argv[2] et la constante **`prod`**. La condition étant vraie, la string **message** est modifiée avec **`the product is`** et **value** est modifiée avec la valeur **1*2*3*4*5** valant **120**.
Ligne 44 : on affiche : **`the product is 120`**

3.3) **`./prog 3 prod`** **`[Y]`**

f() renvoie l'entier **4**
Cette fois-ci **argc** valant **3** une comparaison est effectuée entre la string initialisée avec la chaîne à-la-C argv[2] et la constante **`prod`**. La condition étant vraie, la string **message** est modifiée avec **`the product is`** et **value** est modifiée avec la valeur **1*2*3** valant **6**.
Ligne 44 : on affiche : **`the product is 6`**

3.4) **`./prog 5 operation`** **`[W et Y]`**

f() renvoie l'entier **5**
Cette fois-ci **argc** valant **3** une comparaison est effectuée entre la string initialisée avec la chaîne à-la-C argv[2] et la constante **`prod`**. La condition étant **fausse**, rien n'est fait pour modifier la valeur de **message** ou celle de value qui vaut **5*6/2= 15**
Ligne 44 : on affiche : **`the sum is 15`**

3.4) **`./prog 4 operation`** **`[B]`**

f() renvoie l'entier **4**
Cette fois-ci **argc** valant **3** une comparaison est effectuée entre la string initialisée avec la chaîne à-la-C argv[2] et la constante **`prod`**. La condition étant **fausse**, rien n'est fait pour modifier la valeur de **message** ou celle de value qui vaut **4*5/2= 10**
Ligne 44 : on affiche : **`the sum is 10`**

3.4) **`./prog 3 operation`** **`[S]`**

f() renvoie l'entier **3**
Cette fois-ci **argc** valant **3** une comparaison est effectuée entre la string initialisée avec la chaîne à-la-C argv[2] et la constante **`prod`**. La condition étant **fausse**, rien n'est fait pour modifier la valeur de **message** ou celle de value qui vaut **3*4/2= 6**
Ligne 44 : on affiche : **`the sum is 6`**

4) (6 pts) Compléter le code [All]

Le but du tri pancake est de trier une liste en utilisant le moins d'inversions possible de la liste que l'on veut trier. Le pseudo-code du tri est fourni plus bas (***avec la convention des indices de liste compris entre 1 et la taille de la liste***):

Le pseudocode utilise un algorithme appelé **flip** qui modifie la liste **L** fournie en premier paramètre en inversant tous les éléments d'indices compris entre 1 et le second parametre inclus. Exemple: l'appel **`flip(L,3)`** sur la liste **L** initialisée avec **{10, 20, 30, 14, 15}**, modifie la liste **L** qui prend comme nouvelle valeur **{30, 20, 10, 14, 15}**.

On se sert de **flip()** à 2 endroits dans l'algorithme general du tri pancake comme suit:

```
Algorithme Tri_pancake:
Entrée: entier n > 0
Entrée modifiée: liste L de taille n

Pour k de n à 2 par pas de -1
    // recherche de l'indice du maximum des valeurs de L
    // parmi les k premiers éléments de L
    max_index <- max(L,k)
    // si le maximum n'est pas déjà le dernier element de la sous-liste
    Si max_index != k
        flip(L, max_index) // déplace le maximum au début de la sous-liste
        flip(L, k)         // déplace le maximum en fin de sous-liste
```

Dans cet exercice, le but est de traduire ce pseudocode en C++ puis d'indiquer le résultat de l'exécution de la fonction main() visible ci-dessous:

Ce morceau partiel de programme compile en C++11 sans warning

```
1   #include <iostream>
2   #include <vector>
3   #include <string>
4   using namespace std;
5
6   struct Animal{
7       string espece;
8       Animal* ancetre;
9       int taille_moyenne;
10  };
11
12  void afficheEspeces(vector<Animal*> a);
13  void flip(vector<Animal*>& a, int i);
14  int trouveMax(vector<Animal*> a, int k);
15  void triPancake(vector<Animal*>& a );
16  void afficheAncetres(Animal* p);
17
18  int main()
19  {
20      Animal teck = {"teckel", nullptr, 18};
21      Animal chat = {"chat", nullptr, 25};
22      Animal munc = {"munchkin", &chat, 21};
23      Animal minu = {"minuet", &munc, 22};
24      Animal chev = {"cheval", nullptr, 160};
25      Animal belu = {"beluga", nullptr, 420};
26
27      vector<Animal*> a={&chev, &teck, &munc, &belu, &minu, &chat};
28      triPancake(a);
29      afficheEspeces(a);
30      afficheAncetres(&minu);
31
32      return 0;
33  }
```

4.1) question indépendante : écrire le code de la fonction **afficheEspeces** qui affiche le champ **espece** de chacun des éléments du vector transmis en paramètre (avec un seul nom **d'espece** par ligne). C'est possible en deux lignes de code.

```
34  void afficheEspeces(vector<Animal*> a)
35  {
36      for (auto elem : a)
37          cout << elem->espece << endl;
38  }
```

4.2) question indépendante : écrire le code de la fonction **afficheAncetres** qui affiche **l'espece** de **l'ancetre** du paramètre **p** si un **ancetre** est défini et poursuit l'affichage tant que **l'ancetre** possède lui-même un **ancetre**, etc… (avec un seul nom **d'espece** par ligne). C'est possible en 4 lignes de code.

```
46  void afficheAncetres(Animal* p)
47  {
48      if(p) // recommandé en général mais pas exigé pour cet examen
49          while(p->ancetre != nullptr)
50          {
51              cout << p->ancetre->espece << endl;
52              p = p->ancetre ;
53          }
54  }
55
```

4.3) Ecrire le code de la fonction **trouveMax** qui renvoie l'indice de la valeur maximum du champ **taille_moyenne** présente dans le vector **a** parmi les **k** premiers éléments. C'est possible en 5 lignes de code.

```
64  int trouveMax(vector<Animal*> a, int k)
65  {
66      int mi(0);
67      for (size_t i(1); i < k; ++i)
68          if (a[i]->taille_moyenne > a[mi]->taille_moyenne)
69              mi = i;
70
71
72      return mi;
73  }
```

4.4) Ecrire ensuite le code de la fonction **flip** qui modifie le vector **a** fourni en premier paramètre en inversant tous les éléments d'indices compris entre son premier élément et celui d'indice **i** donnée par le second parametre. C'est possible en 9 lignes de code.

```
82  void flip(vector<Animal*>& a, int i)
83  {
84      Animal* temp(nullptr);
85      int start(0);
86      while (start < i)
87      {
88          temp = a[start];
89          a[start] = a[i];
90          a[i] = temp;
91          start++;
92          i--;
93      }
94  }
95
96
```

4.5) Utiliser les 2 fonctions précédentes et le pseudocode pour écrire la fonction **triPancake** qui modifie le vector **a** en le triant. C'est possible en 5 lignes de code.

```
107 void triPancake(vector<Animal*>& a )
108 {
109     for (size_t k(a.size()); k > 1; --k)
110     {
111         int max_index(trouveMax(a, k));
112         if (max_index != (k-1))
113         {
114             flip(a, max_index);
115             flip(a, k-1);
116         }
117     }
118 }
119
120
```

4.6) le programme est exécuté ; préciser à gauche l'affichage de l'appel de la ligne 29 et à droite l'affichage de l'appel de la ligne 30 :

Affichage de la ligne 29 : selon l'ordre **après le tri** du vector **a** en ligne 28

[W]	[B]	[Y]	[S]
teckel **munchkin** **minuet** **chat** **cheval** **beluga**	**munchkin** **minuet** **chat** **teckel** **cheval** **beluga**	**munchkin** **minuet** **teckel** **chat** **cheval** **beluga**	**munchkin** **teckel** **minuet** **chat** **cheval** **beluga**

Affichage de la ligne 30 : d'abord l'espece de l'ancetre de **minu** puis l'espece de l'ancetre de **munc** et la boucle s'arrete car **chat** n'a pas d'ancetre. **[All]**

munchkin
chat