ICC MT & EL Test C++ /White vendredi 17 décembre 2021

Ne PAS retourner ces feuilles avant d'en être autorisé!

Merci de poser votre carte CAMIPRO en évidence sur la table.

Vous pouvez déjà compléter et lire les informations ci-dessous:

NOM en MAJUSCULE _____

PRENOM _____

Numéro SCIPER _____

Signature _____

BROUILLON : Ecrivez aussi votre NOM-Prénom sur la feuille de brouillon fournie. Toutes vos réponses doivent être sur cette copie d'examen. Les feuilles de brouillon sont ramassées pour être immédiatement détruites.

Le test écrit commence à: **15h15**

Nous recommandons de consacrer 1h20 à l'examen de C++ et 1h40 à l'examen théorique

Les deux copies d'examens sont ramassées à : **17h45**

Le contrôle de ICC reste SANS appareil électronique

Vous avez le droit d'avoir tous vos documents **personnels** sous forme papier: dictionnaire, livres, cours, exercices, code, projet, notes manuscrites, etc...

ECRIVEZ LISIBLEMENT !

Vous pouvez utiliser un crayon à papier et une gomme

Ce contrôle écrit de C++ permet d'obtenir **19 points** sur un total de 100 points pour le cours complet.

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)

```
1   bool single_bit_difference(char octet_a, char octet_b)
2   {
3        unsigned a(octet_a), b(octet_b);
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25  }
```

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  }
```

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.

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

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`**

3.2) **`./prog 10`**

3.3) **`./prog 4 prod`**

3.4) **`./prog 5 operation`**

4) (6 pts) Compléter le code

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
37
38
39
40
41
42
43
44
45  }
```

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
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63  }
```

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
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81  }
```

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
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106 }
```

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
110
111
112
113
114
115
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 :