TESTC++final Sem1 MT & EL 19.01.2021 EPFL / R. Boulic

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

1) (8 pts) structure de contrôle, opérateurs divers et récursivité [All]

Le code suivant compile en C++11 et s'exécute correctement.

```
1   #include <iostream>
2   using namespace std;
3
4   int p ( int n, int k )
5   {
6       if(n<k)
7           return 0;
8       else if( n == k || k == 1 )
9           return 1;
10      else
11          return p( n-1, k-1) + p( n-k, k ) ;
12  }
13
14  int my_function ( int n )
15  {
16      int resultat(0), i(0);
17
18      while ( i++ < n )
19          resultat += p ( n, i );
20
21      return resultat;
22  }
23
24  int main ()
25  {
26      cout << my_function( 4 ) << endl;
27      return 0;
28  }
```

1.1.1) Donner d'abord le **nombre de passages dans la boucle** des lignes 18 et 19

4 passages
i est post-incrémenté. les tests sont faits pour i valant de 0 à 4 et on quitte la boucle pour la valeur 4

1.1.2) Donner la **liste des appels de la fonction p** à la ligne 19 (sans les appels récursifs). Montrer la valeur des arguments. Ex : p(33,124), p(76,21), ….
Remarque : le détails de l'évaluation des appels de p() est pour la question suivante.

p(4, 1), p(4,2), p(4, 3), p(4,4)

1.2) **Evaluer chaque appel de p** identifié à la question précédente. Préciser les résultats intermédiaires, en particulier les appels récursifs ; si le cas se présente, vous pouvez ré-utiliser la valeur des appels récursifs que vous avez déjà expliquée.

- P(4,1) : le test de la ligne 8 est vrai car k==1 : renvoie la valeur **1**

- P(4,2) : renvoie le résultat de la somme de 2 appels récursifs : p(3,1) + p(2,2)
  - P(3,1) : le test de la ligne 8 est vrai car k==1 : renvoie la valeur **1**
  - p(2,2) : le test de la ligne 8 est vrai car n==k : renvoie la valeur **1**
  - p(4,2) renvoie la valeur 2

- P(4,3) : renvoie le résultat de la somme de 2 appels récursifs : p(3,2) + p(1,3)
  - P(3,2) : renvoie le résultat de la somme de 2 appels récursifs : p(2,1) + p(1,2)
    - P(2,1) : le test de la ligne 8 est vrai car k==1 : renvoie la valeur **1**
    - P(1,2) : le test de la ligne 7 est vrai car n<k : renvoie la valeur **0**
    - P(3,2) renvoie la valeur 1
  - p(1,3) : le test de la ligne 7 est vrai car n<k : renvoie la valeur **0**
  - p(4,3) renvoie la valeur **1**

- p(4,4) : le test de la ligne 8 est vrai car n==k : renvoie la valeur **1**

1.3) A partir des résultats de la question précédente, justifier ce qui est affiché par le programme en détaillant les calculs intermédiaires effectués dans **my_function**.

La valeur affichée est la somme des valeurs renvoyées par les 4 appels de la fonction p, c'est-à-dire 1 + 2 + 1 + 1 qui vaut 5.

==========

2) (15 pts) structuration des données avec vector et struct, surcharge des fonctions

Le code fourni pour cet exercice compile en C++11 et s'exécute correctement.

On désire calculer les durées totales de communication avec différents numéros de téléphones en distinguant les appels sortants (*outgoing*) des appels entrants (*incoming*). Pour cela on a défini plusieurs structures dans le code de la page suivante :

- **Date** : mémorise une date du calendrier avec les champs **day, month, year**
- **Call** : mémorise un appel téléphonique avec les champs :
  - **number** : un numéro de téléphone représenté avec le type string
  - **date** : la date de l'appel
  - **duration** : la durée en minutes
  - **direction** : type de l'appel avec un seul caractère
    - sortant : le caractère est `'O'` comme *Outgoing*
    - entrant : le caractère est `'I'` comme *Incoming*
- **Stats** : mémorise les durées pour un numéro de téléphone avec les champs :
  - **number** : un numéro de téléphone
  - **outgoing** : durée en minutes de tous les appels sortants pour ce numéro
  - **incoming**: durée en minutes de tous les appels entrants pour ce numéro

```
1   #include <iostream>
2   #include <string>
3   #include <vector>
4
5   using namespace std;
6
7   struct Date {
8      unsigned int day;
9      unsigned int month;
10     unsigned int year;
11  };
12
13  struct Call {
14     string number;
15     Date date;
16     unsigned int duration;
17     char direction; // either 'I' for Incoming or 'O' for Outgoing
18  };
19
20  struct Stats {
21     string number;
22     unsigned int outgoing;
23     unsigned int incoming;
24  };
25
26  void display(const Date  &date);      // Question 2.1
27  void display(const Call  &call);      // Question 2.1
28  void display(const Stats &stat);      // Question 2.1
29
30  int find_number(const string& num, const vector<Stats>& vec); //Q2.2
31  void process_call(const Call& call, vector<Stats>& vec);      //Q2.3
32
33  int main()
34  {
35     vector<Call> calls ( {
36      {"0771234567", {4,   1, 2021}, 20, 'I'},
37      {"0798765432", {20, 12, 2020}, 43, 'I'},
38      {"0771234567", {3,   1, 2021}, 89, 'O'},
39      {"0771234567", {2,   4, 2020}, 37, 'O'}
40     });
41
42     for (Call c: calls)
43        display(c);
44
45     vector<Stats> stats;
46
47     for (Call c: calls)
48        process_call(c, stats);
49
50     for (Stats s: stats)
51        display(s);
52
53     return 0;
54  }
```

Après la déclaration des structures on trouve la déclaration des prototypes de fonctions aux lignes 26 à 31. Il faudra écrire le code de ces fonctions (questions 2.1 à 2.3). Mais avant cela, prenez le temps d'examiner la fonction principale qui initialise un **vector** de structures **Call** (lignes 35 à 40) puis on trouve 3 boucles et une déclaration:

- Boucle d'**affichage** de chaque structure **Call**

- Déclaration de `stats` le vector de **Stats** qui est initialement vide (ligne 45)
- Boucle de **traitement** de chaque structure **Call** pour construire `stats`
- Boucle d'**affichage** de chaque structure **Stats**

2.1) Ecrire le code des fonctions **display**

2.1.1) La première fonctions **display** (ligne **26**) doit être utilisée par la seconde (ligne **27**) pour produire un affichage comme celui illustré ici pour la première structure **Call** (ligne 36) :

```
Number: 0771234567, date: 4/1/2021, duration: 20, direction: I
```

```
55 // répartir la ou les instructions d'affichage sur
56 // plusieurs lignes pour une bonne lisibilité
57 void display(const Date &date) // environ 2 lignes
58 {
59     cout << date.day   << "/"
60          << date.month << "/" << date.year;
61
62
63 }
64 // Cette fonction doit utiliser la précédente.
65 // Terminer l'affichage par un passage à la ligne
66 void display(const Call &call) // environ 4 lignes
67 {
68     cout << "Number: "<< call.number << ", date: ";
69
70     display(call.date);
71
72     cout << " duration: "   << call.duration
73          << ", direction: " << call.direction
74          << endl;
78 }
```

2.1.2) la dernière fonction **display** (ligne **28**) affiche une structure **Stats** sur une seule ligne dans le terminal et termine par un passage à la ligne comme dans l'exemple illustré ci-dessous :

```
Number: 0771234567, outgoing minutes: 126, incoming minutes: 20
```

```
79 void display(const Stats &stat) // environ 3 lignes
80 {
81     cout << "Number: " << stat.number
82          << ", outgoing minutes: "
83          << stat.outgoing
84          << ", incoming minutes: "
85          << stat.incoming << endl;
86 }
```

2.2) Ecrire le code de la fonction **find_number** qui renvoie ***l'indice*** de l'élément du vector **vec** qui contient un numéro de téléphone égal au paramètre **num**. Si le numéro de téléphone recherché **num** n'est pas dans un des éléments de **vec** alors cette fonction doit renvoyer la valeur entière **-1**. Quatre à huit lignes suffisent pour cette fonction.

```
89  int find_number(const string& num, const vector<Stats>& vec)
90  {
91      for(size_t i(0); i < vec.size(); i++)
92      {
93          if (vec[i].number == num)
94          {
95              return i;
96          }
97      }
98      return -1;
99
100 }
```

2.3) Ecrire le code de la fonction **process_call** qui met à jour le vector **vec** en analysant l'appel **call**. Il faut utiliser la fonction **find_number** écrite à la question précédente pour déterminer si le numéro de **call** est déjà dans un élément du vector **vec**. S'il n'est pas présent il faut ajouter un nouvel élément à **vec**. Dans tous les cas il faut mettre à jour les champs **outgoing** / **incoming**. Une quinzaine de lignes suffisent pour écrire cette fonction.

Rappel : une structure **Stats** mémorise les durées totales de communication avec un numéro de téléphone en distinguant les appels sortants (*outgoing*) des appels entrants (*incoming*). Par exemple l'affichage obtenu pour le numéro 0771234567 du vector calls (lignes 35-40) est visible en 2.1.2).

```
103 void process_call(const Call& call, vector<Stats>& vec)
104 {
105     int place = find_number(call.number, vec);
106
107     if(place == -1)
108     {
109         Stats s{call.number,0,0};
110         if(call.direction == 'O')
111             s.outgoing = call.duration;
112         if(call.direction == 'I')
113             s.incomping= call.duration;
114         vec.push_back(s);
115     }
116     else
117     {
118         if (call.direction == 'O')
119         {
120             vec[place].outgoing += call.duration;
121         }
122         else if(call.direction == 'I')
123         {
124             vec[place].incoming += call.duration;
125         }
126     }
127 }
```

2.4) Supposons que la fonction **process_call** soit déclarée et définie avec cette en-tête :

```
void process_call(const Call& call, vector<Stats> vec)
```

Le code que vous avez écrit à la question 2.3) compile-t-il avec cette en-tête différente ? (oui/non)
OUI

Si oui, l'exécution donne-t-elle le résultat attendu ? Pourquoi ?
Sinon existe-t-il un moyen simple de modifier votre code (sans changer l'en-tête proposée ici) pour obtenir un résultat correct ? (donner un exemple d'instruction modifiée).

On n'obtient pas le résultat attendu car le paramètre de type **vector** est transmis par valeur. Il s'agit donc d'une copie locale qui est modifiée mais sans mettre à jour le **vector** au niveau du code qui appelle cette fonction.

Si on ne peut pas changer l'en-tête alors on ne peut rien faire pour corriger cette situation.

2.5) Supposons que la fonction **process_call** soit déclarée et définie avec cette en-tête:

```
void process_call(const Call* call, vector<Stats>& vec)
```

Le code que vous avez écrit à la question 2.3) compile-t-il avec cette en-tête différente ? (oui/non)
NON car la syntaxe pour utiliser le paramètre **call** n'est plus correcte car c'est maintenant un pointeur.

Si oui, l'exécution donne-t-elle le résultat attendu ? Pourquoi ?
Sinon existe-t-il un moyen simple de modifier votre code (sans changer l'en-tête proposée ici) pour obtenir un résultat correct ? (donner un exemple d'instruction modifiée).

On peut modifier le code pour que le code compile et réalise le but désiré. Il suffit d'utiliser l'opérateur -> au lieu de point. Exemple : remplacer **call.direction** par **call->direction**

2.6) Supposons que la fonction **process_call** soit déclarée et définie avec cette en-tête:

```
void process_call(const Call& call, vector<Stats>* vec)
```

Le code que vous avez écrit à la question 2.3) compile-t-il avec cette en-tête différente ? (oui/non)
NON car la syntaxe pour utiliser le paramètre **vec** n'est plus correcte car c'est maintenant un pointeur.

Si oui, l'exécution donne-t-elle le résultat attendu ? Pourquoi ?
Sinon existe-t-il un moyen simple de modifier votre code (sans changer l'en-tête proposée ici) pour obtenir un résultat correct ? (donner un exemple d'instruction modifiée).

On peut modifier le code pour que le code compile et réalise le but désiré. Il suffit d'utiliser l'opérateur -> au lieu de point mais seulement pour accéder à une méthode.
Exemple : remplacer **vec.push_back(s)** par **vec->push_back(s)**

Par contre pour accéder à un élément du vector, il faut d'abord déréférencer le pointeur avant d'appliquer l'opérateur donnant accès à un élément du vector.
Exemple : remplacer **vec[place].outgoing** par **(*vec)[place].outgoing**

3) (7 pts) évaluation d'expression et boucle for

Le code suivant compile en C++11 et affiche une valeur entière à l'exécution.

```
1   #include <iostream>
2   using namespace std;
3
4   int main ()
5   {
6       int a(0);
7       for(int x(0); !x&&a<=1 ; ++a)
8           a++;
9
10      cout << a << endl;
11      return 0;
12  }
```

3.1) Préciser **comment** les priorités entre opérateurs s'appliquent pour l'évaluation de la condition de la boucle (ligne 7)

Les opérateurs ! et <= sont prioritaires par rapport au ET_Logique &&.

L'expression se décompose en ces sous-expressions **`(!x) && (a <= 1)`**

le résultat de chaque sous-expression donne un booléen

les 2 booléens sont combinés avec le ET_Logique

3.2) combien de passages sont effectués dans cette boucle ? :……1……..

Préciser votre réponse en **évaluant la condition de boucle** à partir de la valeur de **a** et **x** à chaque passage dans la boucle ;

	x	!x	a	a<=1	**(!x) && (a <= 1)**
Valeur initiale	0		0		
Première exécution Ligne 7	0	true	0	true	true
Après exécution Ligne 8			1		
Après exécution ++a (ligne 7)			2		
Nouvelle exécution ligne 7	0	true	2	false	false

3.3) En vous appuyant sur votre réponse à la question précédente, quelle valeur est affichée à la ligne 10 ?

La valeur finale affichée de **a** est **2** car c'est sa valeur quand on quitte la boucle.

4) (8 pts) pointeur et appel de fonction

Le code suivant compile en C++11 et affiche 2 valeurs entières à l'exécution.

```
1  #include <iostream>
2  using namespace std;
3
4  void f2(int* p)
5  {
6      *p = -*p ;
7  }
8
9  int f1(int x)
10 {
11     f2(&x);
12     return  x+1;
13 }
14
15 int main()
16 {
17     int x(3), y(2);
18     y = f1(x);
19     cout << x << " , " << y << endl;
20     return 0;
21 }
```

4.1) Quel est le résultat de l'exécution de la ligne 6 ?

- P est un pointeur sur une variable entière.
- Cette instruction modifie la variable pointée par p grâce à l'opérateur *.
- La nouvelle valeur de la variable pointée par p est l'opposée de l'ancienne valeur de la variable.
- L'analyse de l'appel de la fonction f2 montre que qu'on passe l'adresse du paramètre x de f1.
- C'est ce paramètre dont la valeur va être modifiée.
- Ce paramètre x étant une variable locale à f1, sa nouvelle valeur est ensuite utilisée pour calculer la valeur x+1 qui est renvoyée par f1.
- Il n'y a pas d'impact sur le paramètre x de main.

C'est OK si les éléments de réponse sont fournis pour l'une ou l'autre question

4.2) Donner et justifier l'affichage produit à la ligne 19

- Ligne 18 : c'est un passage par valeur qui est effectué.
- La valeur 3 sert à initialiser le paramètre x de f1.
- Ensuite f2 modifie ce paramètre x de f1 qui prend la valeur -3.
- Enfin f1 renvoie la valeur -3 +1 c'est-à-dire -2 mémorisée dans y

- Ligne 19 : affiche les valeur courante de x et y, c'est-à-dire : 3 , -2