TESTC++final Sem1 MT & EL 05.11.2021 EPFL / R. Boulic

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

1) (4 pts) : Compléter une fonction vérifiant si la valeur absolue d'un entier est un palindrome

Un entier positif est un **palindrome** s'il est symétrique. Par exemple, **414** et **123321** sont des palindromes et **412** et **1231** n'en sont pas. Un entier ayant un seul chiffre est aussi un palindrome. On demande de compléter la fonction **`palindrome`** qui accepte un entier en entrée et qui doit déterminer si *sa valeur absolue* est un **palindrome** ou pas.

1.1) Remplacer les 4 expressions numérotées __(numéro)__ par un opérateur du C++ : [All]

__(1)__ doit être remplacé par l'opérateur : %

__(2)__ doit être remplacé par l'opérateur : *

__(3)__ doit être remplacé par l'opérateur : +

__(4)__ doit être remplacé par l'opérateur : /

```
1   void palindrome(int num)
2   {
3       int n(0), digit(0), rev(0);
4
5       n = num = (num>0)? num : -num ; // valeur absolue
6
7       do
8       {
9           digit = num __(1)__ 10;
10          rev   =(rev __(2)__ 10) __(3)__ digit;
11          num   = num __(4)__ 10;
12      } while (num != 0);
13
14      cout << " The reverse of n is: " << rev << endl;
15
16      if (n == rev)
17          cout << " The number is a palindrome.";
18      else
19          cout << " The number is not a palindrome.";
20  }
```

1.2) on effectue l'appel **`palindrome(718)`** : utiliser le nombre de lignes nécessaires pour indiquer les valeurs intermédiaires des variables **`digit`**, **`rev`** et **`num`** au moment de chaque exécution de la ligne 12 : [W]

passage	digit	rev	num
1	8	8	71
2	1	81	7
3	7	817	0

1.2) on effectue l'appel **`palindrome(817)`** : utiliser le nombre de lignes nécessaires pour indiquer les valeurs intermédiaires des variables **`digit`**, **`rev`** et **`num`** au moment de chaque exécution de la ligne 12 : [B]

passage	digit	rev	num
1	7	7	81
2	1	71	8
3	8	718	0

1.2) on effectue l'appel **`palindrome(518)`** : utiliser le nombre de lignes nécessaires pour indiquer les valeurs intermédiaires des variables **`digit`**, **`rev`** et **`num`** au moment de chaque exécution de la ligne 12 : [Y]

passage	digit	rev	num
1	8	8	51
2	1	81	5
3	5	815	0

1.2) on effectue l'appel **`palindrome(815)`** : utiliser le nombre de lignes nécessaires pour indiquer les valeurs intermédiaires des variables **`digit`**, **`rev`** et **`num`** au moment de chaque exécution de la ligne 12 : [S]

passage	digit	rev	num
1	5	5	81
2	1	51	8
3	8	518	0

2) (4 pts) Surcharge et valeur par défaut

Ce programme compile en C++11 et s'exécute correctement

```
1   #include <iostream>
2   using namespace std;
3
4   char f ( int n )
5   {
6         return (n + 3) % 26 + 'a';
7   }
8
9   int f ( char c )
10  {
11       return 10 * ( c - 'a' );
12  }
13
14  int f ( double d = 1.5 )
15  {
16       int e(d) ;
17       return f(2*e) - 'a' ;
18  }
19
20  int main ()
21  {
22        // Question 2.1
23        cout << f( 'd' ) << endl;
24        cout << f( 5 )   << endl;
25
26        // Question 2.2
27        cout << f ( 10  / 2  )  << endl;
28        cout << f ( 10. / 2  )  << endl;
29        cout << f ( 10  / 2. )  << endl;
30        cout << f ( 10. / 2. )  << endl;
31
32        // Question 2.3
33        cout << "f () :" << f () << endl;
34        return 0;
35  }
```

2.1) justifier l'affichage obtenu pour l'exécution des lignes suivantes : [All]
Ligne 23 : surcharge ligne9 => affiche **30** car 'd'-'a' vaut 3 et que la fonction renvoie 10*3.
Ligne 24 : surcharge ligne4 => affiche **i** car (5+3)%26 + 'a' donne 8+'a' => le code ascii de 'i'

2.2) y a-t-il des différences d'affichage pour l'exécution des lignes 27 à 30 ?
Justifier les éventuelles différences d'affichage.
La ligne 17 est différente car le type de l'expression est **int** donc appel de la surcharge ligne 4
L'argument vaut 5 ; on obtient la même réponse que pour la ligne 24 : affiche **i**

Les lignes 28 à 30 ont toutes un argument de type double qui vaut 5. => surcharge ligne 14
e est initialisé à 5
renvoie **f(10) – 'a'** avec la surcharge ligne 4 pour f(10) qui renvoie **'a'+13**, donc
renvoie **'a' + 13 –'a'** c'est-à-dire **13**

2.3) L'exécution de f() à la ligne 33 produit-elle un affichage ? Justifier la réponse.
Oui, la surcharge ligne 14 est appelée avec la valeur par défaut de 1.5 pour d
Renvoie **f(2) –'a'** c'est-à-dire **'a' + 5 –'a'** qui vaut **5**

3) (4 pts) Entrées-sorties standards

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

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6       int number(0);
7       int sum(0);
8
9       cout << "Enter a number: ";
10      cin  >> number;
11
12      while (number >= 0)
13      {
14          sum += number;
15          cin >> number;
16      }
17      cout << sum  << endl;
18
19      return 0;
20  }
```

Expliquer l'exécution du programme. Indiquer s'il y a *un affichage produit par le programme ou pas*. Si *oui lequel*. **Attention l'exécution est relancée pour chacune des questions de 3.1 à 3.4.**
Remarque : pour chaque cas, on indique la suite des touches du clavier qui ont été frappées. L'indication **Enter** signifie qu'on a frappé une fois la touche de validation (passage à la ligne).

3.1) frappe des touches : [All]

2	0	Enter

Pas d'affichage car le programma attend une lecture en ligne 15

3.2) frappe des touches: [All]

-	1	0	Enter

On n'entre pas dans la boucle car la condition est fausse
Affiche la valeur **0** pour sum car cette variable n'a pas été modifiée

3.3) frappe des touches : [W]

1	5	Enter	1	5	Enter	-	1	5	Enter

On entre dans la boucle 2 fois pour additionner 15 et 15 à sum.
On sort quand -15 est lu. L'affichage final est 30

3.4) frappe des touches (une case vide signifie qu'on a frappé une fois la barre d'espace) : [W]

4	0		4	0		-	2	0	Enter

Même principe que 3.3) car peu importe les séparateurs
On entre dans la boucle 2 fois pour additionner 40 et 40 à sum.
On sort quand -20 est lu. L'affichage final est 80

3.3) frappe des touches : [B]

4	0	Enter	4	0	Enter	-	4	0	Enter

On entre dans la boucle 2 fois pour additionner 40 et 40 à sum.
On sort quand -40 est lu. L'affichage final est 80

3.4) frappe des touches (une case vide signifie qu'on a frappé une fois la barre d'espace) : [B]

1	5		1	5		-	3	0	Enter

Même principe que 3.3) car peu importe les séparateurs
On entre dans la boucle 2 fois pour additionner 15 et 15 à sum.
On sort quand -30 est lu. L'affichage final est 30

3.3) frappe des touches : [Y]

1	0	Enter	1	0	Enter	-	1	0	Enter

On entre dans la boucle 2 fois pour additionner 10 et 10 à sum.
On sort quand -10 est lu. L'affichage final est 20

3.4) frappe des touches (une case vide signifie qu'on a frappé une fois la barre d'espace) : [Y]

1	5		1	5		-	3	0	Enter

Même principe que 3.3) car peu importe les séparateurs
On entre dans la boucle 2 fois pour additionner 15 et 15 à sum.
On sort quand -30 est lu. L'affichage final est 30

3.3) frappe des touches : [S]

1	5	Enter	1	5	Enter	-	1	5	Enter

On entre dans la boucle 2 fois pour additionner 15 et 15 à sum.
On sort quand -15 est lu. L'affichage final est 30

3.4) frappe des touches (une case vide signifie qu'on a frappé une fois la barre d'espace) : [S]

2	0		2	0		-	1	0	Enter

Même principe que 3.3) car peu importe les séparateurs
On entre dans la boucle 2 fois pour additionner 20 et 20 à sum.
On sort quand -10 est lu. L'affichage final est 40

4) (3 pts) Récursivité

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

```
1   #include <iostream>
2   using namespace std;
3
4   void f(char c);
5   int main()
6   {
7       char c('A');
8       cin >> c;
9       f(c);
10      cout << c;
11      return 0;
12  }
13
14  void f(char c)
15  {
16      if (c != '.')
17      {
18          char new_char('A');
19          cin >> new_char;
20          f(new_char);
21          cout << new_char;
22      }
23  }
```

Voici la suite des touches frappées au clavier pour l'exécution de ce programme (l'indication **Enter** signifie qu'on a frappé une fois la touche de validation (passage à la ligne)) : [W]

L	**Y**	**N**	**X**	**.**	**Enter**

4.1) Utiliser le nombre de lignes nécessaires pour indiquer, pour chaque appel récursif, la valeur reçue à la **ligne 14** pour le paramètre formel **c** de la fonction **f**, la valeur de la variable **new_char** après exécution de la **ligne 19** et s'il produit un autre appel récursif (si oui avec quel argument en **ligne 20**).

appel	c Ligne 14	new_char Ligne 19	Autre appel récursif ? si oui, avec quel argument ? Ligne 20
1	L	Y	Y
2	Y	N	N
3	N	X	X
4	X	.	.
5	.		

4.2) S'il y en a un, quel est l'affichage produit dans le terminal ? **.XNYL**

4.3) En bref, que se passe-t-il pour le même input si on inverse les **lignes 20** et **21** du code ?
Y a-t-il un affichage ? Si oui, lequel ? Oui il y a affichage de : **YNX.L**

[B]

W	O	L	F	.	Enter

4.1)

appel	c Ligne 14	new_char Ligne 19	Autre appel récursif ? si oui, avec quel argument ? Ligne 20
1	W	O	O
2	O	L	L
3	L	F	F
4	F	.	.
5	.		

4.2) S'il y en a un, quel est l'affichage produit dans le terminal ? **.FLOW**

4.3) En bref, que se passe-t-il pour le même input si on inverse les **lignes 20** et **21** du code ?
Y a-t-il un affichage ? Si oui, lequel ? Oui il y a affichage de : **OLF.W**

[Y]

Z	E	B	U	.	Enter

4.1)

appel	c Ligne 14	new_char Ligne 19	Autre appel récursif ? si oui, avec quel argument ? Ligne 20
1	Z	E	E
2	E	B	B
3	B	U	U
4	U	.	.
5	.		

4.2) S'il y en a un, quel est l'affichage produit dans le terminal ? **.UBEZ**

4.3) En bref, que se passe-t-il pour le même input si on inverse les **lignes 20** et **21** du code ?
Y a-t-il un affichage ? Si oui, lequel ? Oui il y a affichage de : **EBU.Z**

[S]

L	I	O	N	.	Enter

4.1)

appel	c Ligne 14	new_char Ligne 19	Autre appel récursif ? si oui, avec quel argument ? Ligne 20
1	L	I	I
2	I	O	O
3	O	N	N
4	N	.	.
5	.		

4.2) S'il y en a un, quel est l'affichage produit dans le terminal ? **.NOIL**

4.3) En bref, que se passe-t-il pour le même input si on inverse les **lignes 20** et **21** du code ?
Y a-t-il un affichage ? Si oui, lequel ? Oui il y a affichage de : **ION.L**

---

5) (4 pts) Priorité des opérateurs et évaluation d'expressions

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

```
1   #include <iostream>
2   using namespace std;
3
4   int main ()
5   {
6         int a(1);
7         int j(2 + 2 * --a);
8         cout << " j = " << j << endl;
9         cout << " a = " << a << endl;
10
11        int k(2 + 1 / 2 + 1);
12        cout << " k = " << k << endl;
13
14        bool b(k % 2 == 0);
15        cout << " b = " << b << endl;
16
17        for( int i = 0; i >= 0; i = ( i > 3 ) ? i - 5 : i + 2 )
18        {
19              cout <<  " i = " << i << endl;
20        }
21        return 0;
22  }
```

Compléter la table en précisant l'affichage obtenu à l'exécution de ce programme. Indiquer ce qui est affiché (2ième colonne) et justifier comment cette valeur a été obtenue (colonne de droite), c'est-à-dire ***avec quelle expression et comment cette expression a été évaluée ; indiquer la valeur des sous-expressions s'il y en a***. Ajouter des lignes si nécessaire. [All]

N° Ligne	Affichage	Justification
8	`j = 2`	2 + (2*0) car **a** est décrémentée avant utilisation
9	`a = 0`	Valeur décrémentée en ligne 7
12	`k = 3`	2 + (1/2) + 1 = 2 + 0 + 1 = 3
15	`b = 0`	Comme k vaut 3 alors (k%2) vaut 1 qui est différent de zéro Il y a donc affichage de 0 qui correspond à la valeur de false
19	`i = 0`	La valeur initiale de i produit une condition vraie pour entrer dans la boucle
19	`i = 2`	A la fin du premier passage, i étant inférieur ou égal à 3 on lui ajoute 2. Comme i vaut 2 la condition est vraie et on affiche i=2.
19	`i = 4`	A la fin du second passage, i étant inférieur ou égal à 3 on lui ajoute 2. Comme i vaut 4 la condition est vraie et on affiche i=4.
		A la fin du 3ième passage, i>3 donc on lui retranche 5, ce qui donne la valeur -1. Donc on quitte la boucle car la condition est fausse.