IN 101 - TD 01

Solution

Dans les exercices, lorsqu'il vous est demandé d'écrire une fonction puis de la tester, implicitement vous devrez écrire un `main` qui permet d'appeler et tester cette fonction. Vous pourrez récupérer les entrées nécessaires pour tester la fonction en utilisant `scanf` (vous n'avez pas encore vu comment interagir avec les arguments de ligne de commande).

1 Caractère d'une chaîne

Q 1.1. Écrivez une fonction `nth` prenant en argument une chaîne de caractères et une position et **affichant** le caractère se trouvant à cette position dans la chaîne.

Rajoutez ensuite un `main` qui demande à l'utilisateur d'entrer au clavier une chaîne et une position (en utilisant `scanf`).

Solution

Une solution immédiate peut être :

```c
#include <stdio.h>
#include <stdlib.h>

void nth (char *str, int index) {
  printf ("%c\n", str[index]) ;
}

int main () {
  char str[64] ;
  int pos ;

  printf ("Chaîne et position ? : ") ;
  scanf ("%s %d", str, &pos) ;
  nth (str, pos) ;
  return 0 ;
}
```

Q 1.2. Testez votre programme avec les entrées `"foo"`, 1. Cette chaîne permet-elle de faire un test intéressant ?

Solution

```
$ gcc -Wall nth.c
$ ./a.out
Chaîne et position ? : foo 1
o
```

On affiche bien le caractère `'o'` ce qui est correct. Néanmoins, comme il y a 2 `'o'` consécutifs dans la chaîne, même le programme **incorrect** suivant :

```c
void nth (char *str, int index) {
  printf ("%c\n", str[index + 1]) ;
}
```

aurait **semblé** fonctionner. Il faut donc tester différentes positions et/ou différentes chaînes de caractères.

Q 1.3. Testez votre programme avec `"aloha"`, -10. Que révèle ce test ? Modifiez votre code si nécessaire.

Solution

Si l'on a écrit le programme tel que proposé en Q1.1, on obtient une lettre quelconque. Cela ne provoque pas d'erreur car :

1. `C` ne vérifie pas si l'on accède en dehors des bornes d'un tableau (puisque les bornes ne font pas partie de la structure d'un tableau).
2. Pour des raisons compliquées de compilation, nous ≪avons de la chance≫ : l'accès en dehors du tableau correspond encore à une zone mémoire appartenant au programme. C'est une fausse ≪chance≫ car nous ne voyons pas que notre programme est incorrect.

Quoi qu'il en soit, ce test ne donne pas un résultat significatif. Il faut donc gérer ce **cas particulier** qui est un cas **d'erreur**. Nous pouvons choisir d'afficher un message d'erreur.

```
void nth (char *str, int index) {
  if (index < 0) printf ("Erreur. Indice hors limites.\n") ;
  else printf ("%c\n", str[index]) ;
}
```

```
$ ./a.out
Chaîne et position ? : aloha -10
Erreur. Indice hors limites.
```

Nous pouvons faire mieux en interdisant de manière structurelle la possibilité d'avoir un indice négatif : lui donner le type `unsigned int`. Ainsi, le test devient inutile.

```
void nth (char *str, unsigned int index) {
  printf ("%c\n", str[index]) ;
}
```

Q 1.4. Testez votre programme avec `"aloha"`, 100000. Que révèle ce test ? Modifiez votre code si nécessaire.

Solution

Si l'on a écrit le programme tel que proposé en Q1.3, on obtient :

```
$ ./a.out
Chaîne et position ? : aloha 100000
Segmentation fault: 11
```

car l'index est supérieur **ou égal** à la taille de la chaîne. En `C`, les chaînes sont des tableaux et sont donc indicées de 0 à **taille -1**. Il faut donc vérifier que l'index est compatible avec la taille de la chaîne. La fonction `strlen` de `C` permet d'obtenir la taille d'une chaîne (il faut inclure `string.h`).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void nth (char *str, unsigned int index) {
  if (index >= strlen (str))
    printf ("Erreur. Indice hors limites.\n") ;
  else printf ("%c\n", str[index]) ;
}

int main () {
```

```
  char str[64] ;
  int pos ;

  printf ("Chaîne et position ? : ") ;
  scanf ("%s %d", str, &pos) ;
  nth (str, pos) ;
  return 0 ;
}
```

```
$ ./a.out
Chaîne et position ? : aloha 100000
Erreur. Indice hors limites.
```

Q 1.5. Testez à nouveau votre programme avec `"aloha"`, -10 et interprétez le résultat.

Solution

```
$ ./a.out
Chaîne et position ? : aloha -10
Erreur. Indice hors limites.
```

On obtient le même message que lorsque l'on tente d'accéder au-delà de la taille de la chaîne. En effet, la configuration binaire de l'entier négatif se retrouve interprétée comme un entier positif. De part le codage en complément à 2, le bit de poids fort est à 1, ce qui représente un entier positif très grand ($\geq 2^{31}$).

Q 1.6. On souhaite maintenant que la fonction **retourne** le caractère plutôt qu'elle ne l'affiche. Quel problème se pose à vous ? Proposer des solutions.

Solution

Le problème est de retourner quelque chose lorsque l'on est dans un cas d'erreur, quelque chose que l'on ne confonde pas avec une valeur de retour ≪normale≫, ≪légale≫.

`C` ne dispose d'aucun mécanisme qui permette **d'interrompre** l'exécution d'une fonction **sans retourner** de valeur. Nous devons donc trouver un moyen d'alerter qu'une erreur est survenue.

1. Utiliser une variable globale booléenne disant si le résultat retourné est pertinent. Pour signaler une erreur, on met cette variable à ≪faux≫ et on retourne un caractère quelconque. Cette solution n'est pas satisfaisante car elle ne supporte pas des appels récursifs et nuit à la maintenabilité du programme puisque la fonction modifie des variables en dehors de son corps.
2. Créer une `struct` avec un champ pour le caractère à retourner et un champ booléen pour la validité de ce caractère. L'inconvénient est de devoir déclarer une structure de donnée supplémentaire, juste ≪utilitaire≫. Cela augmente ≪artificiellement≫ la taille du code source (à écrire et à maintenir).
3. Passer un argument supplémentaire booléen, par adresse, dans lequel on transmet à la fonction appelante l'information de validité de la valeur retournée. Cette solution est clairement la meilleure car elle ne présente aucun des inconvénients des solutions précédentes.

2 Polynôme

Q 2.1. Écrivez une fonction `poly` prenant en argument un nombre x et retournant la valeur du polynôme $5x^3 + 2x - 3x^4 + 5$ évalué en ce nombre.

Attention : n'utilisez pas la fonction `pow` de la bibliothèque standard qui masque la réalité abordée dans cet exercice.

Rajoutez ensuite un `main` qui demande à l'utilisateur d'entrer au clavier un réel (flottant) (en utilisant `scanf`).

Solution

La solution immédiate est :

```c
#include <stdio.h>
#include <stdlib.h>

/* 5x^3 + 2x - 3x^4 + 5 -> 11 opérations. */
float poly (float x) {
  return 5 * x * x * x + 2 * x - 3 * x * x * x * x + 5 ;
}

int main () {
  float x ;
  printf ("Point d'évaluation ? : ") ;
  scanf ("%f", &x) ;
  printf ("poly (%f) = %f\n", x, poly (x)) ;
  return 0 ;
}
```

```
$ gcc -Wall poly_naive.c
$ ./a.out
Point d'évaluation ? : 1
poly (1.000000) = 9.000000
$ ./a.out 10
poly (10.000000) = -24975.000000
```

Q 2.2. Combien d'opérations votre programme effectue-t-il ?

Solution

Il y a 8 multiplications et 3 additions / soustractions, ce qui fait 11 opérations.

Q 2.3. Comment pouvez-vous réécrire votre algorithme pour diminuer ce nombre ?

Solution

Si l'on factorise x on peut réécrire le polynôme sous la forme :

$$\begin{aligned}
5x^3 + 2x - 3x^4 + 5 &= \\
-3x^4 + 5x^3 + 2x + 5 &= \\
x\left(-3x^3 + 5x^2 + 2\right) + 5 &= \\
x\left(x\left(-3x^2 + 5x\right) + 2\right) + 5 &= \\
x\left(x\left(x\left(-3x + 5\right)\right) + 2\right) + 5
\end{aligned}$$

On n'a plus que 4 multiplications et 3 additions, donc 7 opérations et le nouvel algorithme plus efficace est :

```c
#include <stdio.h>
#include <stdlib.h>

/* Factorise pour éviter de calculer les même puissances -> 7 opérations. */
float poly (float x) {
  return x * (x * (x * (-3 * x + 5)) + 2) + 5 ;
}
```

```c
int main () {
  float x ;
  printf ("Point d'évaluation ? : ") ;
  scanf ("%f", &x) ;
  printf ("poly (%f) = %f\n", x, poly (x)) ;
  return 0 ;
}
```

Conclusion

L'exercice de réflexion nous a permis d'obtenir un algorithme plus **efficace** car exécutant moins d'opérations.

3 Colinéarité

On souhaite écrire une fonction prenant les coordonnées de 3 points **dans le plan** et **retournant** une valeur disant si ces 3 points sont alignés.

Q 3.1. Quels sont les domaines des entrées et des sorties de cette fonction ?

Solution

La fonction prend 3 fois 2 coordonnées qui sont chacune un **réel**. On peut choisir de prendre en entrée 3 *couples* de réels (encodage avec une `struct`) ou 6 réels. Prenons la dernière solution pour cette correction et nommons les `xa`, `ya`, `xb`, `yb`, `xc`, `yc`.

La fonction doit retourner « oui » ou « non », « vrai » ou « faux ». C'est donc une valeur de **vérité**, c'est donc un **booléen**.

Q 3.2. Sachant que si les points A, B et C sont alignés alors $\overrightarrow{AB}$ et $\overrightarrow{BC}$ sont colinéaires, donc $\overrightarrow{AB} = k\,\overrightarrow{BC}$, imaginez un algorithme et écrivez la fonction `col` qui réalise ce calcul.

Solution

Comme introduit dans la question, si les points A, B et C sont alignés, alors $\overrightarrow{AB}$ et $\overrightarrow{BC}$ sont colinéaires. Donc on a $\overrightarrow{AB} = k\,\overrightarrow{BC}$. D'où

$$x_b - x_a = k\,(x_c - x_b)$$

et

$$y_b - y_a = k\,(y_c - y_b)$$

donc

$$\frac{x_b - x_a}{x_c - x_b} = \frac{y_b - y_a}{y_c - y_b}$$

Il en découle l'algorithme suivant :

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool col (float xa, float ya, float xb, float yb, float xc, float yc) {
  return ((xb - xa) / (xc - xb) == (yb - ya) / (yc - yb)) ;
}

int main () {
  float xa, ya, xb, yb, xc, yc ;
```

```
  printf ("Coordonnées x y de A B C ? : ") ;
  scanf ("%f %f %f %f %f %f", &xa, &ya, &xb, &yb, &xc, &yc) ;
  if (col (xa, ya, xb, yb, xc, yc)) printf ("Vrai\n") ;
  else printf ("Faux\n") ;
  return 0 ;
}
```

Q 3.3. Testez votre programme, regardez par exemple pour `col (3, 3, 3, 3, 3, 3)`.

Solution

Si l'on a écrit le programme comme en Q3.2, on obtient :

```
$ ./a.out
Coordonnées x y de A B C ? : 3 3 3 3 3 3
Faux
```

Effectivement, lorsque des points sont **confondus**, des dénominateurs peuvent être nuls, ce qui provoque une division par 0. Le résultat de la division est alors un flottant particulier (`NaN`, et non une erreur franche) différent de tous les autres flottants. Le test répond alors ≪faux ≫ alors que les points, confondus, sont bien alignés.

Notez que dans certains langages de programmation, la division par 0 est une erreur qui interrompt l'exécution du programme.

Q 3.4. Si l'on a rencontré le problème illustré en Q3.3, réécrire le programme pour l'éviter.

Solution

La première solution est de gérer les 2 cas particuliers de dénominateurs nuls.

```
bool col (float xa, float ya, float xb, float yb, float xc, float yc) {
  float denom1 = (xc - xb) ;
  float denom2 = (yc - yb) ;
  if (denom1 == 0)
    ... bla bla bla return ...
  if (denom2 == 0)
    ... bli bli bli return ...
  return ((xb - xa) / denom1 == (yb - ya) / denom2) ;
}
```

Ceci nécessite l'ajout de 2 cas particuliers à gérer. Si l'on reprend la dernière équation de Q3.2 :

$$\frac{x_b - x_a}{x_c - x_b} = \frac{y_b - y_a}{y_c - y_b}$$

on remarque qu'il suffit de faire disparaître les divisions au profit de multiplications pour **ne plus avoir de cas particuliers** et **ne plus avoir de divisions par 0** :

$$(x_b - x_a)(y_c - y_b) = (y_b - y_a)(x_c - x_b)$$

L'algorithme devient alors :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool col (float xa, float ya, float xb, float yb, float xc, float yc) {
  float left = (xb - xa) * (yc - yb) ;
  float right = (xc - xb) * (yb - ya) ;
  float delta = left - right ;
```

```
  return (delta == 0) ;
}

int main () {
  float xa, ya, xb, yb, xc, yc ;

  printf ("Coordonnées x y de A B C ? : ") ;
  scanf ("%f %f %f %f %f %f", &xa, &ya, &xb, &yb, &xc, &yc) ;

  if (col (xa, ya, xb, yb, xc, yc)) printf ("Vrai\n") ;
  else printf ("Faux\n") ;
  return 0 ;
}
```

```
$ ./a.out
Coordonnées x y de A B C ? : 3 3 3 3 3 3
Vrai
```

Remarque

L'utilisation de la propriété du produit scalaire, qui est nul si deux vecteurs sont colinéaires, nous aurait permis d'obtenir directement la dernière et finale version de l'algorithme.
Pour rappel, le produit scalaire de deux vecteurs $\overrightarrow{AB}\begin{pmatrix} x_1 \\ y_1 \\ z_2 \end{pmatrix}$ et $\overrightarrow{BC}\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix}$ est donné par :

$$\begin{pmatrix} y_1\,z_2 - y_2\,z_1 \\ z1\,x_2 - z2\,x_1 \\ x1\,y2 - x_2\,y_1 \end{pmatrix}$$

Puisque dans cet exercice, on considère des vecteurs dans le plan, les composantes z sont nulles, ce qui se réduit à : $x_1\,y_2 - x_2\,y_1 = 0$.
Donc à $(x_b - x_a)\,(y_c - y_b) - (x_c - x_b)\,(y_b - y_a) = 0$.
Donc à $(x_b - x_a)\,(y_c - y_b) = (x_c - x_b)\,(y_b - y_a)$.

Cela nous a juste fait faire temporairement un petit tour dans un espace à 3 dimensions au lieu des 2 de l'énoncé.

Conclusion

L'exercice de réflexion nous a permis d'obtenir un algorithme se programmant plus **simplement**.

Q 3.5. Une fonction n'est pas toujours appelée avec des constantes. Elle peut faire partie de calculs plus complexes. Dans votre `main`, déclarez une variable `x` valant 13.1 et une variable `y` valant $\sqrt{\texttt{x}}$. Testez votre fonction, en l'appelant avec `(1, 1, y * y, x, 5, 5)`.

Remarque : la fonction $\surd$ en `C` s'appelle `sqrt` et vous devez importer les fonctions mathématiques par : **#include** <math.h>. Sur la ligne de compilation, rajoutez l'option `-lm` en fin de ligne de commande :

```
gcc -Wall col.c -lm
```

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
```

```
bool col (float xa, float ya, float xb, float yb, float xc, float yc) {
  float left = (xb - xa) * (yc - yb) ;
  float right = (xc - xb) * (yb - ya) ;
  float delta = left - right ;
  return (delta == 0) ;
}

int main () {
  float x = 13.1 ;
  float y = sqrt (x) ;
  printf ("%d\n", col (1, 1, y * y, x, 5, 5)) ;
  return 0 ;
}
```

```
$ ./a.out
0
```

Les points sont considérés comme non alignés. Pourtant, $\sqrt{13.1} \times \sqrt{13.1}$ devrait être égal à 13.1 et donc nos 3 points étant confondus, on devrait obtenir ≪vrai ≫ en résultat !

Les calculs avec des flottants sont sujets à des **arrondis**. Il faut donc tester les **égalités** et les **inégalités** ≪à un ϵ près ≫.

Q 3.6. Reprenez votre fonction pour corriger le problème identifié (si vous avez eu le problème) et testez son fonctionnement.

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

#define EPSILON (0.00001)

bool col (float xa, float ya, float xb, float yb, float xc, float yc) {
  float left = (xb - xa) * (yc - yb) ;
  float right = (xc - xb) * (yb - ya) ;
  float delta = left - right ;
  return (-EPSILON < delta && delta < EPSILON) ;
}

int main () {
  printf ("%d\n", col (1, 1, sqrt (11.5) * sqrt (11.5), 11.5, 5, 5)) ;
  return 0 ;
}
```

```
$ ./a.out
1
```

4 Somme à partir de mêmes chiffres

Q 4.1. Écrivez une fonction `same_sum` prenant en argument un entier n et un chiffre c entre 0 et 9 inclus et vérifiant si n est égal à une somme finie de c.

Solution

Si $n = c + ... + c$ alors $\exists p \mid n = p \times c$, donc n est divisible par c, donc n `mod` $c = 0$. Ce raisonnement fonctionne bien pour $n \geq 0$ et $c \neq 0$. Si $c = 0$ alors il y a une division par 0 à cause du modulo.

Récapitulons... Si $n < 0$ alors le résultat est toujours faux puisque une somme de positifs est toujours positive. Si $c = 0$, alors c'est vrai seulement si $n = 0$. Et sinon, modulo. Il y a donc 3 cas dans l'algorithme qui s'écrit comme suit :

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>


bool same_sum (int n, int c) {
  /* Impossible d'avoir une somme négative de chiffres positifs. */
  if (n < 0) return (false) ;
  /* Ne pas diviser par 0. La somme est correcte seulement si elle vaut 0. */
  if (c == 0) {
    if (n == 0) return (true) ;
    else return (false) ;
  }
  else {
    if (n % c == 0) return (true) ;
    else return (false) ;
  }
}

int main () {
  int n, c ;

  printf ("Nombre et chiffre ? : ") ;
  scanf ("%d %d", &n, &c) ;
  printf ("%d\n", same_sum (n, c)) ;
  return 0 ;
}
```

Remarquons que l'écriture de ce code est inutilement compliquée par des imbrications de conditionnelles. Au lieu d'écrire «if condition retourner vrai sinon retourner faux », il est plus court et plus lisible d'écrire «retourner condition ». Le programme se réécrit de manière **totalement équivalente** en :

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>


bool same_sum (int n, int c) {
  /* Impossible d'avoir une somme négative de chiffres positifs. */
  if (n < 0) return (false) ;
  /* Ne pas diviser par 0. La somme est correcte seulement si elle vaut 0. */
  if (c == 0) return (n == 0) ;
  return (n % c == 0) ;  /* Implicitly "else". */
}

int main () {
  int n, c ;

  printf ("Nombre et chiffre ? : ") ;
  scanf ("%d %d", &n, &c) ;
  printf ("%d\n", same_sum (n, c)) ;
  return 0 ;
}
```

Q 4.2. Testez votre programme, par exemple avec same_sum (80, 40). Que doit-il répondre ?

Solution

```
$ ./a.out
Nombre et chiffre ? : 80 40
1
```

Ce test ne respecte clairement pas les **conditions imposées** par l'énoncé puisque 40 n'est pas un chiffre (puisque c'est un nombre) entre 0 et 9.

Le choix de laisser passer **silencieusement** une utilisation erronée est dangereux car le résultat (qui n'a plus aucun sens par rapport au problème posé) va se propager ailleurs dans le programme pour causer éventuellement des défaillances à un **tout autre** endroit. L'identification de la cause de cette défaillance sera difficile puisque tout s'est passé « normalement ».

Pour cette raison, **sauf mention contraire** dans la spécification d'un programme à écrire, il faut couvrir **tout** le domaine informatique de définition des entrées et **lever explicitement** des erreurs dans les cas incorrects ou inattendus.

Q 4.3. Proposez et implémentez une solution.

Solution

Nous avons déjà abordé la question du signalement des erreurs dans l'exercice 1 avec des solutions nécessitant un travail supplémentaire. Si l'on examine le cas présent, on se rend compte que nous ne sommes pas tout à fait dans la même situation : nous ne sommes pas dans le cas où toutes les valeurs du type de retour peuvent être pertinentes. Nous retournons actuellement un booléen, donc 2 valeurs significatives possibles. Nous pouvons créer un type avec 3 valeurs signifiant « vrai », « faux » et « erreur ». Un `enum` est tout à fait adapté. Lors d'un appel à notre fonction, il ne faudra plus tester la valeur retournée comme un booléen, mais discriminer les 3 valeurs possibles obtenues.

Nous pouvons déjà contraindre le type du chiffre à être un `unsigned int` ce qui nous décharge structurellement de la vérification de positivité. Il ne nous reste alors qu'à vérifier que le chiffre est bien < 10.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

/* Un booléen optionnel. */
enum bool_option {
  BO_False, /* Booléen faux. */
  BO_True,  /* Booléen vrai. */
  BO_Error  /* Pas un booléen, mais une erreur à la place. */
};

enum bool_option same_sum (int n, unsigned int c) {
  if (c > 9) return BO_Error ;
  /* Impossible d'avoir une somme négative de chiffres positifs. */
  if (n < 0) return (BO_False) ;
  /* Ne pas diviser par 0. La somme est correcte seulement si elle vaut 0. */
  if (c == 0) return ((n == 0) ? BO_True : BO_False) ;
  return ((n % c == 0) ? BO_True : BO_False) ;  /* Implicitement "else". */
}

int main () {
  int n, c ;

  printf ("Nombre et chiffre ? : ") ;
  scanf ("%d %d", &n, &c) ;

  switch (same_sum (n, c)) {
  case BO_True : printf ("Vrai\n") ; break ;
  case BO_False : printf ("Faux\n") ; break ;
  case BO_Error : printf ("Erreur. Entrée invalide.\n") ; break ;
```

```
  }
  return 0 ;
}
```

Notons que pour éviter de réintroduire une cascade de `if-then-else`, nous avons utilisé **l'expression conditionnelle `?-:`** afin d'alléger l'écriture de la conversion booléen → `enum`.

```
$ ./a.out
Nombre et chiffre ? : 80 4
Vrai
$ ./a.out
Nombre et chiffre ? : 80 3
Faux
$ ./a.out
Nombre et chiffre ? : 80 40
Erreur. Entrée invalide.
```

5 Chaîne d'un nombre + 1

Q 5.1. Écrivez une fonction `changed_digit` prenant en argument 1 chaîne de caractères représentant un nombre écrit en base 10, et dit si le chiffre de poids le plus fort (donc le plus à gauche dans l'écriture) changera si l'on ajoute 1 à ce nombre.

Par exemple, `changed_digit ("123")` renverra ≪faux≫ alors que `changed_digit ("19")` renverra ≪vrai≫.

Attention : on considère par hypothèse que la chaîne représente bien un entier (ne contient pas de caractères illégaux) et n'est donc forcément pas vide. Pas la peine de faire ces vérifications dans votre algorithme.

Attention : on considère qu'il n'y a pas de 0 non significatifs en tête.

Solution

La première solution qui vient à l'esprit est sans doute de calculer la valeur entière que représente cette chaîne. Cela dit, une fois que l'on a cette valeur, il n'y a rien de bien simple pour vérifier que si on lui ajoute 1, son premier chiffre change...À part re-transformer la valeur obtenue après incrémentation en une chaîne et comparer le premier chiffre de chaque chaîne. C'est bien compliqué !

Regardons un peu quand le premier chiffre change. Pour qu'il change, il faut que l'addition de 1 provoque une cascade de retenues remontant jusqu'au chiffre le plus à gauche. Par exemple 2999 qui donnera 3000. On voit donc qu'il faut que tous les chiffres suivant celui le plus à gauche soient des 9. Si le nombre n'a qu'un seul chiffre, alors forcément il va changer. Il reste le cas de la chaîne vide, que l'on ignore par hypothèses de l'exercice. L'algorithme a donc la forme :

```
changed_digit (s) =
  Si s ne contient qu'un seul chiffre, alors retourner vrai
  Vérifier les caractères de la chaîne à partir de l'indice 1 jusqu'à la fin
    Si caractère différent de '9' retourner faux
  Retourner vrai.
```

Comme nous sommes en `C` nous savons que les chaînes sont closes par un `'\0'` final, donc pas besoin de calculer la longueur de la chaîne pour savoir si elle ne contient qu'un seul caractère, ni pour savoir quand arrêter de parcourir ses caractères.

```c
#include <stdio.h>
#include <stdbool.h>

bool change (char *str) {
  if (str[1] == '\0') return true ;
  for (int i = 1; str[i] != '\0'; i++) {
    if (str[i] != '9') return false ;
  }
  return true ;
}


int main () {
  char str[64] ;

  printf ("Chaîne ? : ") ;
  scanf ("%s", str) ;
  printf ("%d\n", change (str)) ;
  return 0 ;
}
```

Q 5.2. Testez votre programme avec `changed_digit ("-10")`. Testez-le avec `changed_digit ("+19")`. Qu'en déduisez-vous ?

Solution

Si l'on a écrit le programme comme ci-dessus, ces tests renvoient incorrectement faux. L'énoncé disait ≪1 chaîne de caractères représentant un nombre écrit en base 10 ≫, pas ≪nombre positif ≫, pas ≪nombre non signé ≫. Notre algorithme est donc incomplet.

Q 5.3. Si besoin, corrigez votre algorithme.

Solution

Pour le cas des nombres négatifs, un raisonnement similaire à celui fait en Q5.1 nous permet de constater que le premier chiffre change si tous les suivants sont des 0. Il faut donc dissocier les deux cas. Il faut également traiter les chaînes commençant par `'+'` pour les gérer comme des entiers positifs sans signe explicite.

```
changed_digit (s) =
  Si s commence par '-'
    Si s ne contient qu'un seul chiffre, alors retourner vrai
    Vérifier les caractères de la chaîne à partir de l'indice 2 à la fin
      Si caractère différent de '0' retourner faux
  Sinon si s commence par '+'
    Si s ne contient qu'un seul chiffre, alors retourner vrai
    Vérifier les caractères de la chaîne à partir de l'indice 2 à la fin
      Si caractère différent de '9' retourner faux
  Sinon
    Si s ne contient qu'un seul chiffre, alors retourner vrai
    Vérifier les caractères de la chaîne à partir de l'indice 1 à la fin
      Si caractère différent de '9' retourner faux
  Retourner vrai.
```

Cet algorithme est un peu répétitif. Il serait bien de pouvoir factoriser tous ces traitements qui se ressemblent. On peut choisir de mémoriser dans une variable quel caractère on doit trouver pour retourner ≪vrai ≫. D'autre part, on peut aussi mémoriser à partir de quel indice

on doit parcourir le reste de la chaîne. Ainsi, on n'aura plus qu'une seule boucle. L'algorithme devient donc :

```
changed_digit (s) =
  Caractère must_be
  Entier start_at = 1
  Si s commence par '-'
     must_be <- '0'
     start_at <- 2
  Sinon si s commence par '+'
     must_be <- '9'
     start_at <- 2
  Sinon si s ne contient qu'un seul chiffre, alors retourner vrai
  Sinon must_be <- '9' (start_at est déjà à 1)
  Tant que l'on n'a pas atteint la fin de la chaîne en partant de start_at
    Vérifier les caractères de la chaîne sont must_be et si non retourner faux
  Retourner vrai
```

```c
#include <stdio.h>
#include <stdbool.h>

bool change (char *str) {
  char must_be ;
  int start_at = 1 ;

  if (str[0] == '-') { must_be = '0' ; start_at = 2 ; }
  else if (str[0] == '+') { must_be = '9' ; start_at = 2 ; }
  else if (str[1] == '\0') return true ;
  else must_be = '9' ;     /* start_at est déjà à 1 par initialisation. */

  while (str[start_at] != '\0') {
    if (str[start_at] != must_be) return false ;
    start_at++ ;
  }
  return true ;
}


int main () {
  char str[64] ;

  printf ("Chaîne ? : ") ;
  scanf ("%s", str) ;
  printf ("%d\n", change (str)) ;
  return 0 ;
}
```

6 Somme de multiples

Q 6.1. Écrivez une fonction `sum` prenant en argument 2 bornes entières `x` et `y` et retournant la somme des nombres **multiples de 5** qui ne sont **pas pairs** entre ces bornes **incluses**.

Solution

L'énoncé n'indique pas que la première borne est nécessairement inférieure à la seconde. Il faut donc commencer à itérer depuis la plus petites des 2 bornes jusqu'à la plus grande.

```c
#include <stdio.h>
#include <stdlib.h>
```

```c
int sum (int x, int y) {
  int accu = 0 ;
  int start = x ;
  int stop = y ;
  int i ;

  /* Vérification de quel argument est le plus petit et on commence à partir
     de lui. */
  if (y < x) {
    start = y ;
    stop = x ;
  }

  for (i = start; i <= stop; i++) {
    if ((i % 5 == 0) && (i % 2 != 0)) accu = accu + i ;
  }
  return accu ;
}

int main () {
  int x, y ;

  printf ("x y ? : ") ;
  scanf ("%d %d", &x, &y) ;
  printf ("%d\n", sum (x, y)) ;
  return 0 ;
}
```

```
$ ./a.out
x y ? : 5 11
5
$ ./a.out
x y ? : 5 15
20
$ ./a.out
x y ? : 4 11
5
$ ./a.out
x y ? : 11 4
5
$ ./a.out
x y ? : 4 21
20
$ ./a.out
x y ? : 4 26
45
$ ./a.out
x y ? : -5 15
15
$ ./a.out
x y ? : 15 -5
15
```

S'il vous reste du temps ou pour continuer après la séance.

7 Différences de paires

Q 7.1. Écrivez une fonction `count` prenant en argument un tableau `t` contenant des entiers, la taille de ce tableau et une valeur entière `n`. Cette fonction doit **afficher** le **nombre** de couples formés d'éléments de `t` dont la différence des composantes vaut `n`.

Le tableau sera directement défini statiquement (≪en dur ≫) dans votre `main` (donc vous connaîtrez sa longueur).

Un squelette de programme (`diffcouples_skel.c`) vous est donné avec un `main` contenant les tests proposés dans la question suivante.

Solution

L'énoncé indique qu'il s'agit de former des **couples**, l'ordre compte donc. Ainsi, $(1, 2)$ est différent de $(2, 1)$. Donc avec le tableau `[1, 2]` et la valeur cible 1, il y a bien un **couple** qui satisfait la condition : $(2, 1)$. Donc pour chaque $(t[i], t[j])$ considéré, il faut vérifier si $t[i]-t[j] = 0$ ou $t[j] - t[i] = 0$.

D'autre part, l'énoncé n'interdit pas de prendre deux fois la même valeur dans le tableau. Cela a un impact dans le cas où la valeur cible est 0 puisqu'il est alors possible de former autant de paires qu'il y a d'éléments dans le tableau, en prenant deux fois le même élément : la différence sera bien égale à 0.

Par contre, attention, dans le cas où l'on prend deux fois le même élément, on ne veut pas compter la combinaison deux fois ($t[i] - t[i]$ et $t[i] - t[i] = 0$).

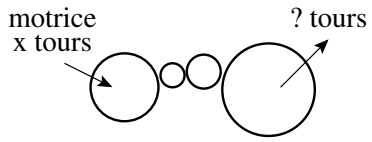
```c
#include <stdio.h>

void count (int *t, unsigned int tlen, int n) {
  int i, j ;
  int cpt = 0 ;

  for (i = 0; i < tlen; i++) {
    /* Le même élément peut être pris 2 fois. */
    if (t[i] - t[i] == n) cpt++ ;
    /* On doit commencer à j égal à la prochaine case. Comme on ne veut
       pas compter 2 fois (à cause de la symétrie testée ci-dessous) le
       cas où on prend la même valeur, on est bien obligé de faire le
       cas particulier ci-dessus. */
    for (j = i + 1; j < tlen; j++) {
      /* Attention, un "couple" est ordonné. Donc (a, b) <> (b, a). */
      if (t[i] - t[j] == n) cpt++ ;
      if (t[j] - t[i] == n) cpt++ ;
    }
  }
  printf ("%d\n", cpt) ;
}


int main () {
#define TLEN (4)
  int t1[TLEN] = { 1, 2, 3, 4 } ;
  int t2[TLEN] = { 0, 0, 0, 0 } ;
  int t3[TLEN] = { -1, -1, -1, -1 } ;

  count (t1, TLEN, 1) ;
  count (t1, TLEN, 0) ;
  count (t1, TLEN, -1) ;
  count (t2, TLEN, 0) ;
  count (t3, TLEN, -1) ;

  return 0 ;
}
```

Q 7.2. Testez votre programme avec au moins les cas suivants et vérifiez si vous obtenez les résultats attendus.

```
count ([1, 2, 3, 4], 1)       → 3
count ([1, 2, 3, 4], 0)       → 4
count ([1, 2, 3, 4], -1)      → 3
count ([0, 0, 0, 0], 0)       → 16
count ([-1, -1, -1, -1], -1)  → 0
```

8 Train d'engrenages

On considère un train d'engrenages (ou roues idéalement collantes) où chaque engrenage est en contact avec un seul suivant. Un tel train est représenté par un tableau contenant le périmètre de chaque engrenage. La première case du tableau contient le périmètre de l'engrenage moteur (que l'on va faire tourner de x tours).



[p1, p2, p3, p4]

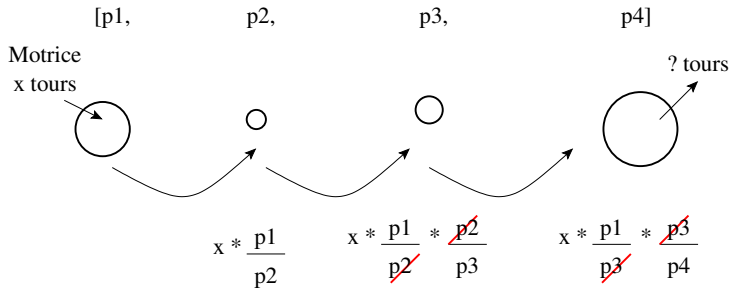
Q 8.1. Écrivez une fonction `turns` prenant en argument un nombre de tours à appliquer au premier engrenage de la chaîne et retournant le nombre de tours effectués par le dernier engrenage de la chaîne.

Solution

Dans une chaîne sans engrenage, on peut considérer que le nombre de tour est trivialement 0. Dans une chaîne à 1 engrenage, le problème est également trivial : c'est la fonction identité.

Dans une chaîne à 2 engrenages, pour 1 tour du premier, le second fait un nombre de tours égal au rapport entre le périmètre du premier engrenage et celui du second. On est ainsi tenté d'itérer sur le tableau en examinant 2 engrenages successifs et appliquant la loi du rapport.

Si l'on analyse ce qui se passe lorsque l'on itère les calculs de nombre de tours engrenage par engrenage dans la chaîne, on remarque que :



les engrenages intermédiaires n'ont aucun impact sur le nombre de tours effectués par l'engrenage final. Seul compte le rapport de diamètre entre le premier et le dernier engrenage, ainsi que le nombre de tours imposés au premier engrenage.

```c
#include <stdio.h>

float turns (float *gears, unsigned int nb_gears, float nb_turns) {
  if (nb_gears == 0) return 0 ;
  if (nb_gears == 1) return nb_turns ;
  return ((nb_turns * gears[0]) / gears[nb_gears - 1]) ;
}

int main () {
#define TLEN (4)
  float t[TLEN] = { 1, 4, 6, 2 } ; /* 1 tour à gauche -> 1/2 tour à droite. */
  printf ("%f\n", turns (t, TLEN, 1.0)) ;
  return 0 ;
}
```

9 Une seule fonction...

Q 9.1. Écrivez un programme qui affiche les arguments de ligne de commande qu'il a reçus.

Attention : pour faire cet exercice, il est indispensable de savoir comment l'on récupère les arguments de la ligne de commande. Si ce n'est pas votre cas, demandez à votre enseignant ou prenez un peu d'avance sur IN102 en lisant le paragraphe du polycopié qui explique ce point.

Contrainte : votre programme **ne devra pas** utiliser de boucles et ne devra comporter que **1 seule** fonction (qui elle, a le droit d'appeler ce qu'elle veut). De même, vous n'avez évidemment pas le droit à l'instruction diabolique `goto` (que nous n'avons pas vue et que nous ne verrons pas car c'est la porte d'entrée à des programmes incompréhensibles et impossibles à maintenir).

Si cela peut vous simplifier, vous pouvez afficher les arguments en ordre inverse de celui sur la ligne de commande.

Solution

Cet exercice a pour but de vous faire prendre conscience que **toutes** les fonctions ont le même statut.

Le programme ne doit contenir qu'une seule fonction. Or `main` est une fonction. Il ne peut donc y avoir d'autre fonction. Mais en C il est possible de faire de la récursion. Si `main` est une fonction, elle peut être récursive, donc s'appeler elle-même. Et la récursivité permet justement d'itérer un traitement : ici ce sera afficher un argument de la ligne de commande.

Il faut donc, à chaque appel récursif fournir un `argc` et un `argv`. Puisque les arguments reçus sur la ligne de commande ne changent pas d'un appel à l'autre, il n'est pas nécessaire de le recréer, on peut utiliser celui reçu à l'appel initial. Pour parcourir la liste de ces arguments, il suffit de décrémenter `argc` et d'arrêter la récursion lorsqu'il est égal à 1. À chaque appel on affichera l'argument se trouvant dans `argv[argc - 1]`.

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
  if (argc == 1) return 0 ;
  else {
    int r = main (argc - 1, argv) ;
    printf ("%s\n", argv[argc - 1]) ;
    return r ;
  }
}
```