I31 – TD4

1. Fonctions

Exercice 1.1.
Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Une fonction permet de factoriser un certain nombre de lignes de code en un endroit du programme. Une fois la fonction définie, elle peut être utilisée autant de fois que nécessaire à partir de n'importe quel autre point du programme (ou presque, nous y reviendrons plus tard). Une fonction accepte un certain nombre de paramètres afin d'appliquer son traitement sur ces valeurs.

En langage C, une fonction est définie de la manière suivante :

type nom([type_param1 param1[, type_param2 param2]*]){
  [intruction ;]*
  [return valeur ;]
}

Une fonction qui ne renvoie rien a pour type de retour `void` et ne nécessite pas de return. Une fonction qui ne prend pas de paramètre peut être définie par :

```
type nom(void){
  [intruction ;]*
  [return valeur ;]
}
```

Ou encore

```
type nom(){
  [intruction ;]*
  [return valeur ;]
}
```

Afin d'être utilisée dans un programme, une fonction doit avoir été déclarée avant sa première utilisation.

Exercice 1.2.
Analyser le programme suivant (donner la séquence des affichages produits) :

```
#include <stdio.h>

int f1(int i){return i+1;}

int f2(int i){return i++;}

int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}

int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}

int main(){
  int a,b;
  a=f1(0);
  b=f2(1);
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Affichage :

```
1 : a=1, b=1
f3 : 0
f4 : 0
2 : a=1, b=0
```

Exercice 1.3.
Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

En langage C, les paramètres d'une fonction sont passés par valeur, c'est-à-dire que la fonction reçoit une copie des valeurs originales de chaque paramètre. Au niveau de la mémoire, lors de l'appel d'une fonction, une pile indépendante est créée puis est détruite à la fin de la fonction sans modifier le programme principal. Ce fonctionnement permet de sécuriser l'exécution mais empêche aussi une fonction de modifier les paramètres et de répercuter ces modifications dans le programme principal.

Exercice 1.4.
Ecrire une fonction int `compare(int x, int y)` qui pour deux nombres *x* et *y* passés en paramètre renvoie *1* si *x* est le plus grand nombre, *-1* si *y* est le plus grand nombre et *0* si les deux nombres sont égaux.

```
int compare(int x, int y) {
  if (x > y){
    return 1;
  } else if (x == y){
    return 0;
  } else {
    return -1;
  }
}
```

Version allégée :

```
int compare(int x, int y) {
  if (x > y){
    return 1;
  }

  if (x == y){
    return 0;
  }

  return -1;
}
```

Exercice 1.5.
Ecrire une fonction float `valeurAbsolue(float x)` qui renvoie la valeur absolue de x.

```
float valeurAbsolue(float x) {
    if (x < 0) {
      return -x;
    }

    return x;
}
```

Exercice 1.6.
Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de *10*. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

Version naïve :

```
int lePlusProcheDe10(int x, int y) {

    int distx = (10-x)*(10-x);

    int disty = (10-y)*(10-y);

    if (distx < disty)
      return x;
    else if (disty < distx)
      return y;
    else
      if (x > y)
        return x;
      else
        return y;
}
```

Version allégée :

```
#include <stdio.h>

int lePlusProcheDe10(int x, int y) {

    int distx = (x > 10)?x - 10:10-x;

    int disty = (y > 10)?y - 10:10-y;

    if (distx < disty) return x;

    if (disty < distx) return y;

    if (x > y) return x;

    return y;
}
```

Exercice 1.7.
Ecrire une fonction int `sommeDesCarres(int n)` qui renvoie la somme des *n* premiers entiers au carré.

```
int sommeDesCarres(int n) {
  int s = 1;

  for(int i = 2; i <=n ; i++)
    s += i*i;

  return s;
}
```

Exercice 1.8.
Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 \leq i \leq j \leq n$.

```
int sommeDesProduits(int n){

    int s = 0;

    for(int i = 1; i <= n; i++){
        for(int j = i; j <= n; j++){
            s += i*j;
        }
    }

    return s;
}
```

Exercice 1.9.
Ecrire une fonction unsigned `int reste(unsigned int x, unsigned int y)` qui renvoie le reste de la division entière de x par y (utiliser une boucle while).

```
int reste(unsigned int x, unsigned int y){

    unsigned int q = 0;         // Le quotient de x/y
    unsigned int r = 0;         // Le reste de x/y

    if (y == 0)                 // Si le diviseur est nul pas de resultat
      return 0;

    r = x;
    while(r >= y){
        r = r - y;

        q += 1;
    }

    return r;
}
```

Exercice 1.10.
Ecrire une fonction void echange(int a, int b) qui échange les valeurs de a et b. Que donne alors le programme suivant :

```
#include <stdio.h>

int main(){
  int a=1,b=2;
  echange(a, b);
  printf("a=%d, b=%d\n",a,b);
}
```

```
void echange(int a, int b){
    int c = a;

    a = b;
    b = c;
}
```

Le programme affiche :

```
a=1, b=2
```

Les paramètres étant passés aux fonctions par valeur (copie), les variables a et b ne sont pas réellement modifiées par le programme.

2. Pointeurs simples

Exercice 2.1.
Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Une variable est un espace dans la mémoire réservé à une valeur d'un certain type. Un pointeur est une variable dont la valeur est une l'adresse d'un espace mémoire (par exemple une autre variable).

En langage C, pour déclarer un pointeur vers une variable d'un certain type on ajoute une * après la déclaration de type. Le tableau suivant illustre les différents types de pointeurs en fonction du type des variables pointées :

Type de variable	Type de pointeur
`char`	`char*`
`short`	`short*`
`int`	`int*`
`float`	`float*`
`double`	`double*`

Il est possible de récupérer l'adresse de n'importe quelle variable en faisant précéder son nom par l'opérateur & (opérateur de référencement). Si une variable est un pointeur, il est possible de récupérer la variable pointée en faisant précéder son nom de l'opérateur * (opérateur de déréférencement).

Pour aller plus loin : https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1042_les-bases-du-langage-c/4286_les-variables/

Exercice 2.2.
Décrire ce que font les instructions suivantes :

```
[1] int i = 10;
[2] int* p;
[3] p = &i;
[4] printf("%p, %d \n", &i, i);
[5] printf("%p, %d \n", p, *p);
```

[1] Déclare l'entier i et l'initialise à 10
[2] Déclare un pointeur vers un entier p
[3] affecte à p l'adresse de la variable i
[4] affiche l'adresse de la variable i puis sa valeur
[5] affiche la valeur de p (qui est l'adresse de i) et déréférence p pour accéder à la valeur pointée (c'est-à-dire celle de i).

Exercice 2.3.
Ecrire un programme qui réalise les actions suivantes :

- Déclarer un entier i initialisé à 0
- Déclarer un pointeur vers un entier p
- Affecter à i une valeur arbitraire
- Faire pointer p vers l'adresse de i
- Saisir une nouvelle valeur pour la variable pointée par p
- Afficher la valeur de i

```
#include <stdio.h>

int main() {
  int i = 0;
  int* p;
  i = 12;
  p = &i;
  scanf("%d", p);
  printf("%d", i);
  return 0;
}
```

Exercice 2.4.
Trouver l'erreur dans cet extrait de programme :

```
int x=5;
int *p = &x;
p = 9;
```

L'instruction p = 9 ne passe pas la compilation car p est de type int* et 9 est de type int. Le type int* représente une adresse et ne peut donc pas être assigné avec un nombre (même si en réalité une adresse est bien un nombre). L'instruction p = (int*)9 corrige le problème mais fait dans ce cas pointer p vers l'adresse 9 de la mémoire, ce qui entrainera très certainement une erreur de segmentation.

Exercice 2.5.
Simplifier les expressions suivantes où p est de type `int*` et i de type `int` :

```
p = &*&i;
i = *&*&j;
```

Simplification:

```
p = &i ;
i = j ;
```

Exercice 2.6.
Refaire l'exercice *1.10*. en écrivant une fonction d'échange permettant réellement d'échanger les 2 valeurs. Modifier le programme principal pour que l'échange soit effectif.

```
void echange(int* a, int* b){
    int c = *a;

    *a = *b;
    *b = c;
}
```

Le programme principal :

```
#include <stdio.h>

int main(){
  int a=1,b=2;
  echange(&a, &b);
  printf("a=%d, b=%d\n",a,b);
}
```