Identification /5

Prénom: .....

Nom: .....

Code permanent: .....

Veuillez choisir **une seule réponse** par question, avec une croix (×) ou un crochet (✓).

Q1. Qu'est-ce qui sera affiché sur la sortie standard par le fragment de code suivant?

```
int a = 7, b = 2;
float f = a / b;
printf("%.1f %d %.1f\n",
       f, a % b, (float)a / b);
```

- ☐ `3.5 1 3.5`
- ☐ `3.0 1 3.0`
- ■ `3.0 1 3.5`
- ☐ `3.5 3 3.5`

Q2. Considérez le fragment de code suivant:

```
char s[20] = "patate";
char t[] = "chou";
strcat(s, t);
```

Une fois ce fragment exécuté, que valent, dans l'ordre, `strlen(s)`, `sizeof(s)` et `sizeof(t)`?

- ☐ `10`, `11` et `4`
- ☐ `10`, `20` et `4`
- ☐ `11`, `20` et `5`
- ■ `10`, `20` et `5`

Q3. Considérez la fonction suivante:

```
void f(int a, int* b) {
  a = a + 1;
  *b = *b + a;
}
```

Qu'est-ce qui sera affiché sur la sortie standard par le fragment de code suivant?

```
int x = 1, y = 2;
f(x, &y);
printf("%d %d\n", x, y);
```

- ■ `1 4`
- ☐ `1 3`
- ☐ `2 4`
- ☐ `2 3`

Q4. La fonction `int main(int argc, char* argv[])` d'un programme `prog` contient l'instruction suivante:

```
printf("%s %c\n", argv[1],
       argv[argc - 1][0]);
```

Qu'est-ce qui sera affiché sur la sortie standard si on lance la commande `./prog -v f.txt 42`?

- ☐ `f.txt 4`
- ■ `-v 4`
- ☐ `-v f`
- ☐ `-v 2`

Q5. Écrivez une fonction C ayant la signature

```
void min_max(const int tab[],
             unsigned int taille,
             int* min, int* max);
```

qui calcule la plus petite et la plus grande valeur du tableau `tab`, de longueur `taille`, et qui les stocke respectivement aux adresses `min` et `max`. Vous pouvez supposer que `taille` vaut au moins `1`. Par exemple, si `tab` contient les valeurs `4`, `-2`, `9` et `0`, alors `-2` doit être stocké à l'adresse `min` et `9`, à l'adresse `max`. *Note:* utilisez le verso au besoin.

Voir explications page suivante

Q1. Les variables `a` et `b` étant toutes deux de type `int`, l'expression `a / b` est une division *entière*, qui vaut `3` (la partie fractionnaire est tronquée). Ce n'est qu'*ensuite* que cette valeur est convertie implicitement en `float` lors de l'affectation à `f`: celle-ci vaut donc `3.0`, et non `3.5`. L'expression `a % b` donne le reste de la division entière de `7` par `2`, soit `1`. Finalement, dans l'expression `(float)a / b`, la conversion explicite s'applique à `a` seulement, puisqu'elle est prioritaire sur la division (voir le tableau de priorité des opérateurs). Un des opérandes étant alors un `float`, l'autre est converti automatiquement et la division est effectuée entre nombres flottants, ce qui donne `3.5`. On obtient donc `3.0 1 3.5`.

Q2. Initialement, le tableau `s` contient la chaîne `"patate"` (6 caractères), suivie du caractère `'\0'`. L'appel `strcat(s, t)` ajoute les 4 caractères de `"chou"` à la fin de `s`, ainsi qu'un nouveau `'\0'`: `s` contient alors `"patatechou"`, de sorte que `strlen(s)` vaut `10`, puisque `strlen` ne compte pas le caractère `'\0'`. Par contre, l'opérateur `sizeof` retourne le nombre d'octets occupés par le tableau lui-même, qui est fixé à la déclaration et évalué à la compilation: `sizeof(s)` vaut donc `20`, peu importe son contenu. Le tableau `t`, dont la taille n'est pas spécifiée, est dimensionné juste assez grand pour contenir la chaîne `"chou"` *et* le caractère `'\0'` final: `sizeof(t)` vaut donc `5` (et non `4`). On remarque aussi que la capacité de `s` (`20`) est suffisante pour contenir le résultat de la concaténation (`11` caractères, en comptant `'\0'`), de sorte qu'il n'y a pas de dépassement de tampon.

Q3. Le premier paramètre de `f` est passé *par valeur*: la fonction reçoit une copie de `x`, de sorte que l'instruction `a = a + 1` modifie la variable locale `a` (qui passe de `1` à `2`), mais pas `x`, qui vaut toujours `1` après l'appel. Le second paramètre est passé *par adresse*: `b` contient l'adresse de `y`, de sorte que l'instruction `*b = *b + a` modifie directement `y`. Comme `a` vaut alors `2`, `y` passe de `2` à `4`. On obtient donc `1 4`.

Q4. Le paramètre `argc` compte tous les arguments, *incluant la commande invoquée*: on a donc `argv[0]` qui vaut `"./prog"`, `argv[1]` qui vaut `"-v"`, `argv[2]` qui vaut `"f.txt"` et `argv[3]` qui vaut `"42"`, de sorte que `argc` vaut `4` (et non `3`). Le premier argument, `argv[1]`, est donc `"-v"`, et non `"f.txt"`: puisque la commande elle-même occupe `argv[0]`, les arguments commencent à l'indice `1`. Finalement, `argv[argc - 1]` désigne le dernier argument, soit `argv[3]`, qui vaut `"42"`: il s'agit d'une chaîne de caractères, et non d'un entier, de sorte que `argv[argc - 1][0]` est son premier caractère (l'indice `0` désigne le premier caractère, et non le dernier), soit `'4'`, affiché avec `%c`. Si on oubliait que `argc` compte la commande, on conclurait à tort que `argc` vaut `3` et que `argv[argc - 1]` est `"f.txt"`, ce qui donnerait `-v f`. On obtient donc `-v 4`.

Q5. Puisqu'une fonction ne peut retourner qu'un seul résultat, les deux valeurs calculées sont transmises au bloc appelant à l'aide de deux pointeurs (passage par adresse), comme dans la fonction `echanger` vue en classe. On initialise d'abord le minimum et le maximum avec le premier élément du tableau, qui existe puisque `taille` vaut au moins `1`. On parcourt ensuite les éléments restants, en mettant à jour le minimum ou le maximum chaque fois qu'on rencontre une valeur plus petite ou plus grande. Deux solutions équivalentes sont présentées ci-dessous: la première travaille directement avec `*min` et `*max`, alors que la seconde utilise des variables locales, qui ne sont copiées aux adresses `min` et `max` qu'à la toute fin.

Version avec `*min` et `*max`

```
void min_max(const int tab[],
             unsigned int taille,
             int* min, int* max) {
  *min = tab[0];
  *max = tab[0];
  for (unsigned int i = 1;
       i < taille; ++i) {
    if (tab[i] < *min)
      *min = tab[i];
    if (tab[i] > *max)
      *max = tab[i];
  }
}
```

Version avec variables locales

```
void min_max(const int tab[],
             unsigned int taille,
             int* min, int* max) {
  int mn = tab[0], mx = tab[0];
  for (unsigned int i = 1;
       i < taille; ++i) {
    if (tab[i] < mn) mn = tab[i];
    if (tab[i] > mx) mx = tab[i];
  }
  *min = mn;
  *max = mx;
}
```

Erreurs fréquentes. Il faut bien distinguer le pointeur `min` de la valeur pointée `*min`: écrire `min = tab[i]` (sans l'astérisque) tente d'affecter un entier à un pointeur, ce qui est interdit, alors que la comparaison `tab[i] < min` compare un entier à une adresse, ce qui n'a pas de sens. Il ne faut pas non plus initialiser le minimum et le maximum à `0`: si toutes les valeurs du tableau sont strictement positives, le minimum obtenu serait alors `0`, une valeur qui n'apparaît pas dans le tableau (et de façon symétrique pour le maximum si toutes les valeurs sont strictement négatives). Finalement, on ne peut pas déterminer la longueur du tableau à l'aide de `sizeof(tab)` dans la fonction: puisque `tab` y est un pointeur, `sizeof(tab)` retourne la taille d'une adresse, et non celle du tableau. C'est précisément pourquoi la longueur est passée explicitement dans le paramètre `taille`.

Utilisation. Pour appeler la fonction, on passe les adresses de deux variables du bloc appelant à l'aide de l'opérateur `&`:

```
int tab[] = {4, -2, 9, 0};
int mn, mx;
min_max(tab, 4, &mn, &mx);
printf("%d %d\n", mn, mx); // Affiche -2 9
```