**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?

```
char c = '7';
int n = c - '0';
double x = n + 0.5;
printf("%d %.1f %c\n",
       n, x, c + 1);
```

☐ `7 7.5 56`
☐ `7 7.0 8`
■ `7 7.5 8`
☐ `55 55.5 8`

**Q2**. Qu'est-ce qui sera affiché sur la sortie standard par le fragment de code suivant?

```
char s[] = "chat", t[] = "chat";
printf("%d %d %d %d\n", s == t,
       strcmp(s, t) == 0,
       strcmp(s, "chaton") < 0,
       strcmp(s, "chameau") < 0);
```

☐ `1 1 1 0`
■ `0 1 1 0`
☐ `0 1 1 1`
☐ `0 1 0 0`

**Q3**. La fonction `int main(int argc, char* argv[])` d'un programme `prog` contient l'instruction suivante:

```
printf("%d [%s] %zu\n", argc,
       argv[2], strlen(argv[2]));
```

Qu'est-ce qui sera affiché sur la sortie standard si on lance la commande `./prog -n 'Hello World' 3`?

■ `4 [Hello World] 11`
☐ `5 [Hello] 5`
☐ `4 ['Hello World'] 13`
☐ `3 [3] 1`

**Q4**. Considérez la fonction suivante:

```
int g(int* p, int q) {
  *p = *p * 2;
  q = q * 2;
  return *p + q;
}
```

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

```
int x = 3, y = 4;
int z = g(&x, y);
printf("%d %d %d\n", x, y, z);
```

☐ `6 8 14`
☐ `6 4 10`
☐ `3 4 14`
■ `6 4 14`

**Q5**. Écrivez une fonction C ayant la signature

```
void compter_signes(
  const int tab[],
  unsigned int taille,
  unsigned int* num_pos,
  unsigned int* num_neg);
```

qui compte le nombre de valeurs strictement positives et le nombre de valeurs strictement négatives du tableau `tab`, de longueur `taille`, et qui stocke ces nombres respectivement aux adresses `num_pos` et `num_neg`. Par exemple, si `tab` contient les valeurs `4`, `-2`, `0`, `9` et `-7`, alors `2` doit être stocké à l'adresse `num_pos` et `2`, à l'adresse `num_neg` (la valeur `0` n'est ni positive, ni négative). *Note:* utilisez le verso au besoin.

**Voir explications page suivante**

**Q1**. Un caractère est représenté par un entier (son code ASCII), de sorte que l'expression `c - '0'` est une soustraction entière entre les codes de `'7'` (55) et de `'0'` (48). Comme les chiffres sont consécutifs dans la table ASCII, on obtient la *valeur* du chiffre, soit `7` (et non son code `55`). Dans l'expression `n + 0.5`, un des opérandes est de type `double`: l'entier `n` est donc converti automatiquement en `double` avant l'addition, et le résultat est `7.5` (la partie fractionnaire n'est pas perdue, puisque le calcul n'est pas fait entre entiers). Finalement, dans l'expression `c + 1`, le caractère `c` est promu en `int`, et on obtient le code suivant celui de `'7'`, soit `56`, qui est le code de `'8'`. Puisque cette valeur est affichée avec `%c`, c'est le caractère correspondant qui est affiché, soit `8`, et non l'entier `56` (qui serait affiché avec `%d`). On obtient donc `7 7.5 8`.

**Q2**. Les tableaux `s` et `t` contiennent la même chaîne, mais occupent deux zones distinctes en mémoire. Dans l'expression `s == t`, chaque tableau se comporte comme un pointeur vers son premier élément: on compare donc deux *adresses*, et non le contenu des chaînes, de sorte que l'expression vaut `0` (d'ailleurs, `gcc` émet un avertissement pour cette comparaison). C'est pourquoi on doit utiliser la fonction `strcmp` pour comparer des chaînes: elle retourne `0` si les chaînes sont égales, une valeur négative si la première précède la seconde dans l'ordre lexicographique, et une valeur positive sinon. Ainsi, `strcmp(s, t) == 0` vaut `1`. La chaîne `"chat"` est un préfixe de `"chaton"`: lorsque les 4 premiers caractères sont égaux, on compare le caractère `'\0'` de `s` avec le caractère `'o'`, et comme `'\0'` a le plus petit code, `"chat"` précède `"chaton"`, de sorte que la troisième expression vaut `1`. Finalement, la comparaison se fait caractère par caractère, de gauche à droite, jusqu'à la première différence, et non selon la longueur des chaînes: les chaînes `"chat"` et `"chameau"` ont les mêmes 3 premiers caractères, puis on compare `'t'` avec `'m'`. Comme `'t'` vient après `'m'`, `"chat"` vient après `"chameau"` (comme dans un dictionnaire), même si elle est plus courte, de sorte que `strcmp(s, "chameau")` est positif et que la dernière expression vaut `0`. On obtient donc `0 1 1 0`.

**Q3**. Avant de lancer le programme, le shell découpe la ligne de commande en arguments selon les espaces, *sauf* ceux qui sont protégés par des apostrophes (ou des guillemets). Le texte `'Hello World'` forme donc un seul argument, et les apostrophes elles-mêmes sont retirées par le shell: elles ne sont pas transmises au programme. On a donc `argv[0]` qui vaut `"./prog"`, `argv[1]` qui vaut `"-n"`, `argv[2]` qui vaut `"Hello World"` et `argv[3]` qui vaut `"3"`, de sorte que `argc` vaut `4` (la commande invoquée est comptée). La longueur de `argv[2]` est `11`: 5 lettres pour `Hello`, 1 espace et 5 lettres pour `World` (le caractère `'\0'` n'est pas compté par `strlen`). Si on oubliait que les apostrophes protègent l'espace, on obtiendrait `argc` égal à `5` et `argv[2]` égal à `"Hello"`; si on croyait que les apostrophes étaient conservées, on obtiendrait une longueur de `13`. On obtient donc `4 [Hello World] 11`.

**Q4**. Le premier paramètre de `g` est passé *par adresse*: `p` contient l'adresse de `x`, de sorte que l'instruction `*p = *p * 2` modifie directement `x`, qui passe de `3` à `6`. Le second paramètre est passé *par valeur*: `q` est une copie de `y`, de sorte que l'instruction `q = q * 2` modifie la variable locale `q` (qui passe de `4` à `8`), mais pas `y`, qui vaut toujours `4` après l'appel. Par contre, la valeur retournée est calculée à partir de la copie locale `q`, qui a bel et bien été modifiée à l'intérieur de la fonction: la fonction retourne donc `6 + 8`, soit `14` (et non `6 + 4`). On obtient donc `6 4 14`.

**Q5**. Puisqu'une fonction ne peut retourner qu'un seul résultat, les deux nombres calculés sont transmis au bloc appelant à l'aide de deux pointeurs (passage par adresse), comme dans la fonction `echanger` vue en classe. On initialise d'abord les deux compteurs à `0`, puis on parcourt le tableau en incrémentant le compteur approprié selon le signe de chaque valeur; les valeurs nulles ne modifient aucun compteur. Deux solutions équivalentes sont présentées ci-dessous: la première travaille directement avec `*num_pos` et `*num_neg`, alors que la seconde utilise des variables locales, qui ne sont copiées aux adresses `num_pos` et `num_neg` qu'à la toute fin.

### Version avec `*num_pos` et `*num_neg`

```
void compter_signes(
  const int tab[],
  unsigned int taille,
  unsigned int* num_pos,
  unsigned int* num_neg) {
  *num_pos = 0;
  *num_neg = 0;
  for (unsigned int i = 0;
       i < taille; ++i) {
    if (tab[i] > 0)
      ++*num_pos;
    else if (tab[i] < 0)
      ++*num_neg;
  }
}
```

### Version avec variables locales

```
void compter_signes(
  const int tab[],
  unsigned int taille,
  unsigned int* num_pos,
  unsigned int* num_neg) {
  unsigned int p = 0, n = 0;
  for (unsigned int i = 0;
       i < taille; ++i) {
    if (tab[i] > 0) ++p;
    else if (tab[i] < 0) ++n;
  }
  *num_pos = p;
  *num_neg = n;
}
```

*Erreurs fréquentes.* Il faut bien distinguer le pointeur `num_pos` de la valeur pointée `*num_pos`: écrire `num_pos = 0` (sans l'astérisque) modifie la copie locale du pointeur, et non le compteur du bloc appelant. Il faut aussi faire attention à la priorité des opérateurs: l'expression `*num_pos++` incrémente le *pointeur* (l'opérateur `++` suffixe est prioritaire sur `*`), et non la valeur pointée. Il faut plutôt écrire `++*num_pos`, `(*num_pos)++` ou `*num_pos += 1`. Il ne faut pas non plus oublier d'initialiser les compteurs à `0`: la fonction ne peut pas supposer que les variables du bloc appelant ont été initialisées. 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, 0, 9, -7};
unsigned int np, nn;
compter_signes(tab, 5, &np, &nn);
printf("%u %u\n", np, nn); // Affiche 2 2
```