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 t[] = {5, 10, 15, 20, 25, 30};
int *p = &t[4], *q = t;
q += 2;
printf("%d %d %d %d\n", *q, p[1],
       (int)(p - q), *(p - 1) + 1);
```

- ☐ `15 30 8 21`
- ■ `15 30 2 21`
- ☐ `15 26 2 21`
- ☐ `15 30 2 25`

Q2. Considérez le programme `joindre.c` suivant:

```
int main(int argc, char *argv[]) {
  if (argc < 3) {
    fprintf(stderr,
            "usage: joindre A B\n");
    return 1;
  }
  printf("%s-%s\n", argv[2], argv[1]);
  return 0;
}
```

Une fois le programme compilé, qu'est-ce qui sera affiché dans le terminal par les commandes suivantes?

```
$ ./joindre un deux > sortie.txt
$ echo $?
```

- ☐ `deux-un`, puis `0`
- ■ `0` seulement
- ☐ `un-deux`, puis `0`
- ☐ `usage: joindre A B`, puis `1`

Q3. Considérez la fonction suivante:

```
int compter(void) {
  static int n = 0;
  int k = 0;
  ++n;
  ++k;
  return 10 * n + k;
}
```

Que retourneront trois appels successifs à `compter()`, dans l'ordre?

- ☐ `11 11 11`
- ☐ `10 20 30`
- ☐ `11 22 33`
- ■ `11 21 31`

Q4. Qu'affichera le fragment de code suivant?

```
struct Etudiant {
  char nom[10]; int note;
};

int compare(const void *a,
            const void *b) {
  const struct Etudiant *x = a, *y = b;
  return y->note - x->note;
}
// ...
struct Etudiant t[] = {
  {"Alice", 72}, {"Bob", 85},
  {"Carl", 64}, {"Dina", 91}};
qsort(t, 4, sizeof(struct Etudiant),
      compare);
printf("%s %s\n", t[0].nom, t[3].nom);
```

- ■ `Dina Carl`
- ☐ `Alice Dina`
- ☐ `Carl Dina`
- ☐ `Bob Carl`

Q5. Complétez le corps de la fonction suivante, qui retourne l'élément du tableau `t`, de taille `n >= 1`, pour lequel `cle` retourne la plus grande valeur (le premier, en cas d'égalité):

```
int maximum_selon(
  const int *t, unsigned int n,
  int (*cle)(int)) {
  // TODO
}
```

Exemple: si `absolue` retourne la valeur absolue de son argument, alors

```
int v[] = {3, -7, 5};
maximum_selon(v, 3, absolue); // -7
```

Note: utilisez le verso.

Voir explications page suivante

Q1. Après l'instruction `q += 2`, le pointeur `q` pointe vers `t[2]`, qui vaut `15`, alors que `p` pointe vers `t[4]`, qui vaut `25`. L'expression `*q` vaut donc `15`. L'expression `p[1]` est équivalente à `*(p + 1)`, soit `t[5]`, qui vaut `30`, et non `*p + 1` (qui vaudrait `26`). La différence `p - q` donne le nombre d'*éléments* qui séparent les deux pointeurs, soit `4 - 2 = 2`, et non le nombre d'octets (qui serait `8`): l'arithmétique des pointeurs tient compte de la taille du type pointé. Finalement, `*(p - 1)` désigne `t[3]`, qui vaut `20`, auquel on ajoute `1`: on obtient `21`, et non `*(p - 1 + 1)`, qui vaudrait `25`. On obtient donc `15 30 2 21`.

Q2. La commande `./joindre un deux` transmet deux arguments au programme: `argc` vaut donc `3` (le nom du programme compte aussi), et la condition `argc < 3` est fausse. Le programme affiche alors `argv[2]`, puis `argv[1]`, soit `deux-un`, sur la sortie standard (`stdout`). Or, la sortie standard est redirigée vers le fichier `sortie.txt` par `>`: ce texte n'apparaît donc pas dans le terminal, mais dans le fichier. Le programme retourne ensuite `0`, ce qui signale une terminaison normale. Cette valeur est conservée dans la variable spéciale `$?`, que la commande `echo $?` affiche. Seul `0` est donc affiché dans le terminal, alors que le fichier `sortie.txt` contient `deux-un`.

Q3. La variable `n` est *statique*: elle est réservée et initialisée à `0` une seule fois, au début du programme, et elle conserve sa valeur d'un appel à l'autre. À l'inverse, la variable locale `k` est automatique: elle est créée et réinitialisée à `0` à chaque appel, et vaut donc toujours `1` au moment du `return`. Le premier appel retourne `10 * 1 + 1 = 11`, le deuxième, `10 * 2 + 1 = 21` et le troisième, `10 * 3 + 1 = 31`. Si `k` était aussi statique, on obtiendrait `11 22 33`, et si aucune des deux variables n'était statique, on obtiendrait `11 11 11`. Les appels retournent donc `11`, `21` et `31`.

Q4. La fonction `qsort` trie un tableau selon la relation d'ordre définie par `compare`: celle-ci doit retourner une valeur négative si le premier élément doit être placé avant le second, une valeur positive s'il doit être placé après, et `0` s'ils sont équivalents. Puisque `qsort` est générique, `compare` reçoit des pointeurs de type `const void*`, qu'on convertit (implicitement) en `const struct Etudiant*` pour accéder aux champs avec l'opérateur `->`. Ici, la fonction calcule `y->note - x->note` plutôt que `x->note - y->note`: le résultat est négatif lorsque `x` a la note la plus *élevée*. Le tableau est donc trié par ordre *décroissant* de note: `Dina` (91), `Bob` (85), `Alice` (72), puis `Carl` (64). Le tri est effectué sur place: c'est le tableau `t` lui-même qui est modifié. On affiche le premier et le dernier élément, soit `Dina Carl`.

Q5. On parcourt le tableau en conservant le meilleur élément trouvé jusqu'ici, qu'on initialise avec le premier élément (ce qui est permis, puisque `n >= 1`). Un pointeur de fonction s'appelle exactement comme une fonction ordinaire: on compare donc `cle(t[i])` à `cle(meilleur)`. La comparaison stricte (`>`) garantit qu'en cas d'égalité, c'est le premier élément rencontré qui est conservé.

```
int maximum_selon(const int *t, unsigned int n, int (*cle)(int)) {
  int meilleur = t[0];
  for (unsigned int i = 1; i < n; ++i)
    if (cle(t[i]) > cle(meilleur))
      meilleur = t[i];
  return meilleur;
}
```

Erreurs fréquentes. Il faut retourner l'*élément* lui-même, et non la valeur de sa clé: dans l'exemple, la fonction doit retourner `-7`, et non `7`. Il ne faut pas non plus initialiser le maximum à `0`: si toutes les clés étaient négatives (par exemple, avec une fonction qui retourne l'opposé de son argument), aucun élément ne serait jamais retenu. Finalement, comparer `cle(t[i])` à `meilleur` (au lieu de `cle(meilleur)`) mélange une clé et un élément, ce qui donne un résultat erroné dès que `cle` n'est pas l'identité.

Variante. Pour éviter de rappeler `cle` sur le meilleur élément à chaque itération, on peut conserver sa clé dans une seconde variable:

```
int maximum_selon(const int *t, unsigned int n, int (*cle)(int)) {
```

```
  int meilleur = t[0], cle_max = cle(t[0]);
  for (unsigned int i = 1; i < n; ++i) {
    int c = cle(t[i]);
    if (c > cle_max) {
      meilleur = t[i];
      cle_max = c;
    }
  }
  return meilleur;
}
```