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[] = {10, 20, 30, 40, 50};
int *p = t + 1, *q = t + 4;
printf("%d %d %d %d\n", *(p + 1),
       q[-1], (int)(q - p), *p + 1);
```

- ☐ `30 40 12 21`
- ☐ `40 40 3 30`
- ■ `30 40 3 21`
- ☐ `30 50 3 21`

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

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr,
            "usage: salut NOM\n");
    return 2;
  }
  printf("Bonjour %s\n", argv[1]);
  return 0;
}
```

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

```
$ ./salut Alice Bob 2> erreurs.txt
$ echo $?
```

- ■ `2` seulement
- ☐ `usage: salut NOM`, puis `2`
- ☐ `Bonjour Alice Bob`, puis `0`
- ☐ `Bonjour Alice`, puis `0`

Q3. Considérez la fonction suivante:

```
int suivant(int pas) {
  static int total = 10;
  total += pas;
  return total;
}
```

Que retourneront les appels `suivant(1)`, `suivant(2)` et `suivant(3)`, dans cet ordre?

- ☐ `11 12 13`
- ■ `11 13 16`
- ☐ `1 3 6`
- ☐ `10 12 15`

Q4. Qu'affichera le fragment de code suivant?

```
struct Pays { char code[3]; int pop; };

int compare(const void *a,
            const void *b) {
  const struct Pays *x = a, *y = b;
  return strcmp(x->code, y->code);
}
// ...
struct Pays t[] = {{"BE", 12},
  {"CA", 40}, {"FR", 68}, {"JP", 124}};
struct Pays cle = {"FR", 0};
struct Pays *p = bsearch(&cle, t, 4,
  sizeof(struct Pays), compare);
printf("%d %d\n",
       (int)(p - t), p->pop);
```

- ☐ `2 0`
- ■ `2 68`
- ☐ `3 68`
- ☐ `68 2`

Q5. Complétez le corps de la fonction suivante, qui retourne le nombre d'éléments du tableau `t`, de taille `n`, pour lesquels `predicat` retourne `true`:

```
unsigned int nombre_si(
  const int *t, unsigned int n,
  bool (*predicat)(int)) {
  // TODO
}
```

Exemple: si `est_pair` retourne `true` lorsque son argument est pair, alors

```
int v[] = {1, 2, 3, 4, 6};
nombre_si(v, 5, est_pair); // 3
```

Note: utilisez le verso au besoin.

Voir explications page suivante

Q1. Le pointeur `p` pointe vers `t[1]` (qui vaut `20`) et `q`, vers `t[4]` (qui vaut `50`). L'expression `*(p + 1)` désigne l'élément suivant celui pointé par `p`, soit `t[2]`, qui vaut `30`. L'expression `q[-1]` est équivalente à `*(q - 1)`, soit `t[3]`, qui vaut `40`: c'est permis, puisque `q - 1` pointe toujours dans le tableau. La différence `q - p` donne le nombre d'*éléments* qui séparent les deux pointeurs, soit `4 - 1 = 3`, et non le nombre d'octets (qui serait `12`): l'arithmétique des pointeurs tient compte de la taille du type pointé. Finalement, le déréférencement est prioritaire sur l'addition: `*p + 1` vaut `(*p) + 1`, soit `21`, et non `*(p + 1)`. On obtient donc `30 40 3 21`.

Q2. La commande `./salut Alice Bob` transmet deux arguments au programme: `argc` vaut donc `3` (le nom du programme compte aussi), et la condition `argc != 2` est vraie. Le message d'utilisation est écrit sur la sortie d'erreur (`stderr`), qui est redirigée vers le fichier `erreurs.txt` par `2>`: il n'apparaît donc pas dans le terminal. Le programme retourne ensuite `2`, une valeur différente de `0` qui signale une terminaison anormale. Cette valeur est conservée dans la variable spéciale `$?`, que la commande `echo $?` affiche. Seul `2` est donc affiché dans le terminal, alors que le fichier `erreurs.txt` contient `usage: salut NOM`.

Q3. La variable `total` est *statique*: elle est réservée et initialisée à `10` une seule fois, au début du programme, et elle conserve sa valeur d'un appel à l'autre. L'initialisation `static int total = 10;` n'est donc pas exécutée à chaque appel. Le premier appel retourne `10 + 1 = 11`, le deuxième, `11 + 2 = 13` et le troisième, `13 + 3 = 16`. Sans le mot réservé `static`, `total` serait réinitialisée à `10` à chaque appel, et on obtiendrait plutôt `11 12 13`. Les appels retournent donc `11`, `13` et `16`.

Q4. La fonction `bsearch` effectue une fouille binaire dans un tableau *trié* selon la relation d'ordre définie par `compare`. Ici, les pays sont comparés selon leur *code*, avec `strcmp` (ordre lexicographique), et le tableau `t` est bien trié selon ce critère: `"BE" < "CA" < "FR" < "JP"`. Puisque `bsearch` est générique, `compare` reçoit des pointeurs de type `const void*`, qu'on convertit (implicitement) en `const struct Pays*` pour accéder aux champs avec l'opérateur `->`. La clé recherchée est elle-même une structure, dont seul le champ `code` est utilisé par `compare`: la valeur `0` du champ `pop` n'a aucune importance. La fonction `bsearch` retourne un pointeur vers l'élément *du tableau* qui correspond à la clé, soit `&t[2]`, et non vers `cle` (elle retournerait `NULL` si aucun élément ne correspondait). La différence `p - t` donne donc la position `2` (les indices commencent à `0`), et `p->pop` vaut `t[2].pop`, soit `68`. On obtient donc `2 68`.

Q5. On parcourt le tableau et on incrémente un compteur chaque fois que `predicat` retourne `true`. Un pointeur de fonction s'appelle exactement comme une fonction ordinaire: `predicat(t[i])`. Le tableau est reçu par un pointeur en lecture seule (`const int *`), puisque la fonction n'a pas à le modifier.

```
unsigned int nombre_si(const int *t, unsigned int n, bool (*predicat)(int)) {
  unsigned int compte = 0;
  for (unsigned int i = 0; i < n; ++i)
    if (predicat(t[i]))
      ++compte;
  return compte;
}
```

Erreurs fréquentes. Il faut écrire `predicat(t[i])`, et non `predicat(i)`: on veut tester la *valeur* de chaque élément, et non son indice. Finalement, il ne faut pas oublier d'initialiser le compteur à `0`: une variable locale non initialisée a une valeur indéterminée.

Variante. On peut aussi parcourir le tableau avec un pointeur plutôt qu'avec un indice:

```
unsigned int nombre_si(const int *t, unsigned int n, bool (*predicat)(int)) {
  unsigned int compte = 0;
  for (const int *p = t; p < t + n; ++p)
    if (predicat(*p))
      ++compte;
  return compte;
}
```