Identification /5

Prénom: .....

Nom: .....

Code permanent: .....

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

Q1. Sur une architecture 64 bits où un `int` occupe 4 octets et un `char`, 1 octet, qu'est-ce qui sera affiché par le fragment de code suivant?

```
char *s, c = 'x';
int n = 5, *p = &n;
double d, *q = &d;
printf("%zu %zu %zu %zu\n", sizeof s,
       sizeof c, sizeof *p, sizeof *q);
```

- ☐ `8 8 4 8`
- ☐ `1 1 4 8`
- ■ `8 1 4 8`
- ☐ `8 1 8 8`

Q2. Considérez le fragment de code suivant:

```
int a = 1, b = 2;
int *const p = &a;
const int *q = &b;
```

Quelle instruction, placée après ce fragment, provoque une **erreur de compilation**?

- ☐ `*p = 3;`
- ☐ `q = &a;`
- ■ `p = &b;`
- ☐ `b = *q + 1;`

Q3. Soit `x` une variable de type `int` et `p` une variable de type `int*` contenant l'adresse de `x`. Parmi les affirmations suivantes, laquelle est vraie?

- ☐ L'affectation `&x = p;` est valide, puisque `&x` désigne l'emplacement de `x`
- ■ Après l'initialisation `int *r = p;`, l'instruction `*r = 7;` modifie la valeur de `x`
- ☐ L'expression `*x` est valide, puisque `x` est une *left-value*
- ☐ Si `q` est de type `const int*`, alors `int *r = q;` est acceptée sans avertissement

Q4. Qu'est-ce qui sera affiché sur la sortie standard par le fragment de code suivant?

```
struct Point { int x; int y; };
struct Point a = {1, 2}, b = a;
struct Point *p = &a, *q = &b;
b.x = 4;
p->y = q->x + 1;
q = p;
q->x = (*p).y * 2;
printf("%d %d %d %d\n",
       a.x, a.y, b.x, b.y);
```

- ☐ `1 5 10 2`
- ☐ `4 5 4 2`
- ☐ `10 5 10 5`
- ■ `10 5 4 2`

Q5. La structure suivante permet de représenter une durée en minutes et en secondes:

```
struct Duree {
  unsigned int minutes;
  unsigned int secondes;
};
```

Écrivez une fonction C ayant la signature

```
void ajouter_duree(
  struct Duree *d,
  const struct Duree *e);
```

qui ajoute la durée pointée par `e` à la durée pointée par `d`, de telle sorte que le champ `secondes` de `d` demeure compris entre `0` et `59`. Vous pouvez supposer que c'est déjà le cas, pour les deux durées, avant l'appel, que le champ `minutes` n'est soumis à aucune contrainte (il peut dépasser `59`) et qu'il n'y a pas de débordement (*overflow*) à gérer. Si `d` ou `e` est `NULL`, alors la fonction ne fait rien. Par exemple, si `d` pointe vers la durée `{2, 50}` (2 minutes et 50 secondes) et `e`, vers la durée `{1, 15}`, alors `d` pointe ensuite vers la durée `{4, 5}`. *Note:* utilisez le verso au besoin.

Voir explications page suivante

Q1. Dans la déclaration `char *s, c = 'x';`, le symbole `*` s'applique uniquement à la variable qui le suit: `s` est un pointeur vers un caractère, alors que `c` est un simple `char`. C'est d'ailleurs pourquoi on préfère coller l'astérisque au nom de la variable (`char *s`) plutôt qu'au type. Ainsi, `sizeof s` vaut `8`, puisque tous les pointeurs occupent le même espace, peu importe le type pointé (même si un `char` n'occupe lui-même qu'un octet), alors que `sizeof c` vaut `1`. Ensuite, `*p` désigne la valeur pointée par `p`, soit l'entier `n`: `sizeof *p` vaut donc `4`, et non `8`, qui serait plutôt la valeur de `sizeof p`. De la même façon, `*q` est de type `double`, de sorte que `sizeof *q` vaut `8` (c'est une coïncidence si cette valeur est la même que celle de `sizeof q`). Notons que `d` n'est pas initialisée, mais ce n'est pas un problème, puisque `sizeof` est évalué à la compilation, à partir du *type* de l'expression seulement. On obtient donc `8 1 4 8`.

Q2. Il faut bien distinguer les deux positions possibles du qualificatif `const`. Le pointeur `p`, déclaré `int *const p`, est un pointeur *constant*: on peut écrire la valeur pointée, mais on ne peut pas modifier `p` lui-même (le faire pointer ailleurs). À l'inverse, le pointeur `q`, déclaré `const int *q`, est un pointeur *en lecture seule*: on peut lire la valeur pointée et on peut modifier `q` lui-même, mais on ne peut pas écrire la valeur pointée *à travers* `q`. Ainsi, `*p = 3;` est valide (on modifie la valeur pointée par `p`, soit `a`), `q = &a;` est valide (on modifie `q`) et `b = *q + 1;` est valide (on lit la valeur pointée par `q` et on modifie `b` directement, qui n'est pas déclaré `const`, même si `q` pointe vers lui). Par contre, `p = &b;` tente de modifier un pointeur constant, ce que le compilateur refuse:

```
error: assignment of read-only variable 'p'
```

Notons que l'instruction qui aurait été refusée pour `q` est plutôt `*q = 3;`.

Q3. L'opérateur `&` retourne l'adresse d'une *left-value*, mais cette adresse est elle-même une *right-value* (une valeur, sans espace mémoire réservé): elle ne peut donc pas être placée à gauche de l'opérateur `=`, et `&x = p;` est refusé par le compilateur (`lvalue required as left operand of assignment`). L'opérateur de déréférencement `*` ne s'applique qu'à un pointeur: puisque `x` est de type `int`, l'expression `*x` est invalide, même si `x` est une *left-value*. La conversion de `const int*` vers `int*` est une conversion vers un pointeur *moins qualifié*: elle permettrait d'écrire, à travers `r`, une valeur qui devait être en lecture seule, de sorte que le compilateur émet un avertissement (`initialization discards 'const' qualifier`). Seule la conversion inverse, de `int*` vers `const int*`, est faite implicitement. Finalement, l'initialisation `int *r = p;` copie l'adresse contenue dans `p`: `r` pointe donc aussi vers `x`, de sorte que `*r` désigne l'espace mémoire de `x` lui-même et que `*r = 7;` stocke `7` dans `x`. C'est donc la deuxième affirmation qui est vraie.

Q4. L'initialisation `b = a` *copie* les champs de `a` dans `b`: les deux structures occupent des espaces mémoire distincts, et modifier l'une ne modifie pas l'autre. Initialement, `p` pointe vers `a` et `q`, vers `b`. L'instruction `b.x = 4;` donne `b` égal à `{4, 2}`, sans modifier `a`. L'instruction `p->y = q->x + 1;` est équivalente à `a.y = b.x + 1;`, de sorte que `a` vaut alors `{1, 5}`. L'instruction `q = p;` modifie ensuite le pointeur `q`, qui pointe désormais vers `a` (et non plus vers `b`). Dans la dernière affectation, `q->x` désigne donc `a.x` et `(*p).y` vaut `a.y`, soit `5`: on stocke donc `10` dans `a.x`. Si on oubliait que `q` pointe désormais vers `a`, on modifierait plutôt `b.x`, ce qui donnerait `1 5 10 2`; si on croyait que `b = a` liait les deux structures, on obtiendrait plutôt `10 5 10 5`. On obtient donc `10 5 4 2`.

Q5. Puisqu'on reçoit des pointeurs vers les structures, et non des copies, les modifications faites dans la fonction sont visibles dans le bloc appelant. La durée `e` est seulement lue: elle est donc reçue à l'aide d'un pointeur en lecture seule (`const`), ce qui permet au compilateur de détecter une écriture non souhaitée. On vérifie d'abord les pointeurs `NULL`, *avant* tout accès aux champs, puisque l'expression `d->secondes` ne doit pas être évaluée si `d` ne pointe vers aucune structure. On additionne ensuite les minutes et les secondes séparément, puis on reporte les secondes en trop dans les minutes. Comme les deux champs `secondes` sont inférieurs à `60`, leur somme est inférieure à `120`: il y a donc au plus une minute à reporter. Par exemple, pour `{2, 50}` et `{1, 15}`, on obtient `3` minutes et `65` secondes, soit `4` minutes et `5` secondes. Deux solutions

équivalentes sont présentées ci-dessous: la première utilise la division entière et le reste (opérateur `%`), alors que la seconde utilise une conditionnelle.

Version avec division et reste

```
void ajouter_duree(
  struct Duree *d,
  const struct Duree *e) {
  if (d == NULL || e == NULL)
    return;
  unsigned int total = d->secondes
                     + e->secondes;
  d->minutes += e->minutes + total / 60;
  d->secondes = total % 60;
}
```

Version avec une conditionnelle

```
void ajouter_duree(
  struct Duree *d,
  const struct Duree *e) {
  if (d == NULL || e == NULL)
    return;
  d->minutes += e->minutes;
  d->secondes += e->secondes;
  if (d->secondes >= 60) {
    d->minutes += 1;
    d->secondes -= 60;
  }
}
```

Erreurs fréquentes. Il faut bien distinguer le pointeur de la structure pointée: l'expression `d.secondes` est invalide, puisque `d` n'est pas une structure, mais un pointeur; il faut écrire `d->secondes` (ou, de façon équivalente mais moins lisible, `(*d).secondes`). Attention également à la priorité des opérateurs: `*d.secondes` est interprété comme `*(d.secondes)`, et non comme `(*d).secondes`. Il faut aussi faire attention à l'ordre des instructions, par exemple dans le fragment suivant:

```
d->secondes = (d->secondes + e->secondes) % 60;
d->minutes += e->minutes + (d->secondes + e->secondes) / 60;
```

La seconde instruction utilise la *nouvelle* valeur de `d->secondes`, ce qui donne un mauvais résultat (pour l'exemple, on obtiendrait `{3, 5}` au lieu de `{4, 5}`). C'est pourquoi on stocke le total dans une variable locale dans la première solution. Finalement, il ne faut pas oublier de reporter les secondes en trop: additionner simplement les champs deux à deux peut donner un champ `secondes` supérieur à `59` (pour l'exemple, on obtiendrait `{3, 65}`).

Utilisation. Pour appeler la fonction, on passe les adresses des structures à l'aide de l'opérateur `&`:

```
struct Duree d = {2, 50}, e = {1, 15};
ajouter_duree(&d, &e);
printf("%u:%02u\n", d.minutes, d.secondes); // Affiche 4:05
```