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?

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

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

Q2. Considérez le fragment de code suivant:

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

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

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

Q3. 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 = {3, 4};
struct Point *p = &a, *q = p;
p->x = 5;
q = &b;
q->y = p->y + (*q).x;
printf("%d %d %d %d\n",
       a.x, a.y, b.x, b.y);
```

- ☐ `1 2 3 5`
- ☐ `5 2 3 7`
- ☐ `5 7 3 4`
- ■ `5 2 3 5`

Q4. 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'expression `&(x + 1)` est valide
- ☐ L'initialisation `double *d = p;` est acceptée sans avertissement
- ■ L'expression `*p` est une *left-value*: on peut donc écrire `*p = 3;`
- ☐ Si `s` est de type `char*`, alors `const char *t = s;` requiert une conversion explicite

Q5. Considérez la structure suivante:

```
struct Compte {
  unsigned int numero;
  double solde;
};
```

Écrivez une fonction C ayant la signature

```
bool transferer(struct Compte *source,
                struct Compte *dest,
                double montant);
```

qui transfère `montant` du compte pointé par `source` vers celui pointé par `dest`, puis retourne `true`. Toutefois, si un des pointeurs est `NULL`, si les deux pointeurs désignent le même compte, si `montant` n'est pas strictement positif ou si le solde de `source` est insuffisant, alors aucun compte n'est modifié et la fonction retourne `false`. Par exemple, si `source` a un solde de `100.0` et `dest`, un solde de `20.0`, alors un transfert de `30.0` donne les soldes `70.0` et `50.0`. *Note:* utilisez le verso au besoin.

Voir explications page suivante

Q1. Dans la déclaration `int *p, q;`, le symbole `*` s'applique uniquement à la variable qui le suit: `p` est un pointeur vers un entier, alors que `q` est un simple `int`. C'est d'ailleurs pourquoi on préfère coller l'astérisque au nom de la variable (`int *p`) plutôt qu'au type. Ainsi, `sizeof p` vaut `8`, alors que `sizeof q` vaut `4`. Puisqu'un pointeur contient une adresse, tous les pointeurs occupent le même espace, peu importe le type pointé: `sizeof r` vaut donc aussi `8`, même si un `double` occupe lui-même 8 octets (c'est une coïncidence). Finalement, `*s` désigne la valeur pointée par `s`, soit le caractère `c`, de type `char`: comme rappelé dans l'énoncé, `sizeof *s` vaut donc `1` (en fait, la norme C garantit que `sizeof(char)` vaut toujours `1`), et non `8`, qui serait plutôt la valeur de `sizeof s`. On obtient donc `8 4 8 1`.

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

```
error: assignment of read-only location '*p'
```

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

Q3. Initialement, `p` et `q` pointent tous les deux vers `a`. L'instruction `p->x = 5;` est équivalente à `(*p).x = 5;`: elle modifie directement le champ `x` de `a`, et non une copie, de sorte que `a` vaut alors `{5, 2}`. L'instruction `q = &b;` modifie ensuite le pointeur `q`, qui pointe désormais vers `b` (sans que `p` soit affecté: l'initialisation `*q = p` copiait simplement l'adresse contenue dans `p`, sans créer de lien entre les deux pointeurs). Dans la dernière affectation, `p->y` vaut donc `a.y`, soit `2`, et `(*q).x` vaut `b.x`, soit `3`: on stocke donc `5` dans `b.y`. Si on oubliait que `q` pointe désormais vers `b`, on modifierait plutôt `a.y`, ce qui donnerait `5 7 3 4`. On obtient donc `5 2 3 5`.

Q4. L'opérateur `&` ne s'applique qu'à une *left-value*, c'est-à-dire une expression ayant un espace mémoire réservé: `x + 1` est une *right-value* (une valeur temporaire, sans adresse), de sorte que `&(x + 1)` est refusé par le compilateur. L'initialisation `double *d = p;` affecte un `int*` à un `double*`: les types sont incompatibles, ce qui produit un avertissement (et même une erreur par défaut depuis GCC 14). Le fait que tous les pointeurs occupent le même espace ne change rien: le type pointé sert justement à détecter ce genre d'erreur à la compilation. La conversion de `char*` vers `const char*` est au contraire une conversion vers un pointeur du même type, mais *plus qualifié*: elle est donc faite *implicitement*, sans qu'aucune conversion explicite ne soit nécessaire. Finalement, `*p` désigne l'espace mémoire de `x` lui-même: c'est donc une *left-value*, et `*p = 3;` stocke `3` dans `x`. C'est donc la troisième affirmation qui est vraie.

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. Avant tout accès, on vérifie les cas où le transfert doit être refusé: on doit tester les pointeurs `NULL` *en premier*, puisque l'expression `source->solde` ne doit pas être évaluée si `source` ne pointe vers aucune structure. L'évaluation paresseuse (*short-circuit*) de l'opérateur `||` garantit que les conditions suivantes ne sont pas évaluées dès qu'une condition est vraie. Pour détecter que les deux pointeurs désignent le même compte, on compare directement les *adresses* avec `==`. Deux solutions équivalentes sont présentées ci-dessous.

Version avec une seule condition

```
bool transferer(struct Compte *source,
                struct Compte *dest,
                double montant) {
  if (source == NULL || dest == NULL
      || source == dest
      || montant <= 0
      || source->solde < montant)
    return false;
  source->solde -= montant;
  dest->solde += montant;
  return true;
}
```

Version avec conditions séparées

```
bool transferer(struct Compte *source,
                struct Compte *dest,
                double montant) {
  if (source == NULL || dest == NULL)
    return false;
  if (source == dest)
    return false;
  if (montant <= 0)
    return false;
  if (source->solde < montant)
    return false;
  source->solde -= montant;
  dest->solde += montant;
  return true;
}
```

Erreurs fréquentes. Il faut bien distinguer le pointeur de la structure pointée: l'expression `source.solde` est invalide, puisque `source` n'est pas une structure, mais un pointeur; il faut écrire `source->solde` (ou, de façon équivalente mais moins lisible, `(*source).solde`). Attention également à la priorité des opérateurs: `*source.solde` est interprété comme `*(source.solde)`, et non comme `(*source).solde`. Pour vérifier que les deux comptes sont distincts, il faut comparer les pointeurs (`source == dest`), et non les numéros ou les soldes: deux comptes distincts peuvent très bien avoir le même solde. Il ne faut pas non plus modifier un des comptes avant d'avoir vérifié *toutes* les conditions, sans quoi un transfert refusé pourrait laisser les comptes dans un état incohérent. Finalement, il ne faut pas oublier d'inclure l'en-tête `<stdbool.h>` pour utiliser le type `bool` (ce n'est plus nécessaire depuis C23, où `bool`, `true` et `false` sont des mots réservés).
Utilisation. Pour appeler la fonction, on passe les adresses de deux structures à l'aide de l'opérateur `&`:

```
struct Compte c1 = {1, 100.0}, c2 = {2, 20.0};
transferer(&c1, &c2, 30.0); // Retourne true
printf("%.1f %.1f\n", c1.solde, c2.solde); // Affiche 70.0 50.0
transferer(&c1, &c1, 10.0); // Retourne false
transferer(&c2, &c1, 60.0); // Retourne false (solde insuffisant)
```