QCM - Pointeurs

Exercices - Chapitre 3 - Solutions [CSC-43043-EP] 2026

Solutions

Question 1. Réponse : **B)**

L'adresse mémoire de la variable

L'opérateur `&` appliqué à une variable retourne son adresse mémoire, c'est-à-dire la position de son premier octet en mémoire vive. Par exemple, `&a` donne l'adresse de la variable `a`. Cette adresse peut être stockée dans un pointeur du type correspondant. C'est l'opérateur fondamental pour obtenir un pointeur vers une variable existante.

Question 2. Réponse : **C)**

```
int*
```

La variable `p` est déclarée comme `int*`, c'est-à-dire un pointeur vers un entier. Elle contient l'adresse mémoire de la variable `x`. Le symbole `*` dans la déclaration indique qu'il s'agit d'un pointeur. La notation `int*` se lit ”pointeur vers int”.

Question 3. Réponse : **C)**

Il accède à la valeur stockée à l'adresse contenue dans le pointeur

L'opérateur `*` appliqué à un pointeur (déréférencement) permet d'accéder à la valeur stockée à l'adresse contenue dans le pointeur. Par exemple, si `p` pointe vers une variable contenant `42`, alors `*p` vaut `42`. Cet opérateur permet à la fois de lire et de modifier la valeur pointée.

Question 4. Réponse : **C)**

```
int* p = nullptr;
```

En C++ moderne (à partir de C++11), `nullptr` est le mot-clé recommandé pour initialiser un pointeur nul. Contrairement à `NULL` (macro C) ou `0` (valeur entière), `nullptr` est de type `std::nullptr_t` et ne peut pas être confondu avec un entier. Ne pas initialiser un pointeur (`int* p;`) est dangereux car il contient une adresse aléatoire.

Question 5. Réponse : **B)**

10

Le pointeur `p` contient l'adresse de `a`. L'instruction `*p = 10` modifie la valeur à l'adresse pointée par `p`, c'est-à-dire la variable `a` elle-même. Donc `a` vaut désormais `10`. C'est le principe fondamental du déréférencement : modifier `*p` revient à modifier directement `a`.

Question 6. Réponse : **C)**

```
delete p;
```

La mémoire allouée avec `new` (pour un seul objet) doit être libérée avec `delete`. Il ne faut pas utiliser `delete[]` qui est réservé aux tableaux alloués avec `new[]`. De même, `free` est la fonction C correspondant à `malloc` et ne doit pas être mélangée avec `new`. La règle est : `new` correspond à `delete`, `new[]` correspond à `delete[]`.

Question 7. Réponse : **B)**

```
delete[] p;
```

Un tableau alloué avec `new[]` doit être libéré avec `delete[]`. Utiliser `delete` sans les crochets sur un tableau provoque un comportement indéfini. La correspondance est stricte : `new` avec `delete`, et `new[]` avec `delete[]`. Mélanger ces opérateurs peut entraîner une corruption mémoire ou des fuites.

Question 8. Réponse : **B)**

Sur la pile (stack)

Les variables locales d'une fonction sont stockées sur la pile (stack). Elles sont automatiquement créées à l'entrée dans le bloc et détruites à la sortie. La pile a une taille limitée (quelques Mo en général). C'est pourquoi on utilise l'allocation dynamique (tas) pour les structures volumineuses.

Question 9. Réponse : **C)**

La pile a une taille limitée (quelques Mo) tandis que le tas peut utiliser plusieurs Go

La pile a une taille limitée, typiquement de l'ordre de 8 Mo par défaut, tandis que le tas peut utiliser plusieurs gigaoctets de mémoire. La pile est gérée automatiquement par le compilateur (allocation et libération des variables locales), alors que le tas nécessite une gestion manuelle ou l'utilisation de pointeurs intelligents. C'est cette différence qui justifie l'allocation dynamique pour les grandes structures.

Question 10. Réponse : **B)**

Un pointeur qui pointe vers une zone mémoire déjà libérée ou invalide

Un pointeur dangling (pointeur pendu) est un pointeur qui contient l'adresse d'une zone mémoire qui a déjà été libérée ou qui n'est plus valide. Cela arrive par exemple quand on retourne l'adresse d'une variable locale, ou quand on utilise un pointeur après un `delete`. Accéder à un pointeur dangling provoque un comportement indéfini. Pour limiter ce risque, on met le pointeur à `nullptr` après libération.

Question 11. Réponse : **C)**

30

Le pointeur `p` pointe vers le premier élément du tableau `tab`. L'expression `*(p + 2)` avance de 2 éléments (pas 2 octets) et déréférence, ce qui donne `tab[2]` soit `30`. C'est le principe de l'arithmétique des pointeurs : `p + N` avance de `N * sizeof(int)` octets. Les expressions `tab[i]` et `*(tab + i)` sont strictement équivalentes.

Question 12. Réponse : **B)**

`const int* p` : on ne peut pas modifier la valeur pointée ; `int* const p` : on ne peut pas modifier l'adresse stockée dans `p`

`const int* p` signifie que la valeur pointée est constante : on ne peut pas faire `*p = 5`. En revanche, on peut changer l'adresse : `p = &autre_variable`. À l'inverse, `int* const p` signifie que le pointeur lui-même est constant : on ne peut pas faire `p = &autre`, mais on peut modifier la valeur pointée via `*p = 5`. Pour un pointeur doublement constant, on utilise `const int* const p`.

Question 13. Réponse : **A)**

`10 30`

Le pointeur `p` pointe d'abord vers `a`, puis est réaffecté pour pointer vers `b`. L'instruction `*p = 30` modifie donc la valeur pointée par `p`, c'est-à-dire `b`, qui passe de 20 à 30. La variable `a` n'est jamais modifiée et reste à 10. L'affichage est donc `10 30`. Réaffecter un pointeur (`p = &b`) change l'adresse stockée, pas la valeur à l'ancienne adresse.

Question 14. Réponse : **B)**

Fuite mémoire (memory leak)

La fonction alloue un entier sur le tas avec `new`, mais ne le libère jamais avec `delete`. Quand la fonction se termine, le pointeur local `p` est détruit (il était sur la pile), mais la mémoire allouée sur le tas persiste sans possibilité d'y accéder. C'est une fuite mémoire classique. Si cette fonction est appelée en boucle, la consommation mémoire croît indéfiniment.

Question 15. Réponse : **B)**

8 octets

Sur une machine 64 bits, un pointeur occupe 8 octets (64 bits), quel que soit le type pointé. Que ce soit un `int*`, un `double*`, un `char*` ou un `void*`, la taille du pointeur est toujours de 8 octets car il doit pouvoir stocker une adresse mémoire de 64 bits. Sur une machine 32 bits, un pointeur occuperait 4 octets.

Question 16. Réponse : **B)**

`*(tab + i)`

L'expression `tab[i]` est strictement équivalente à `*(tab + i)` en C et C++. Le compilateur convertit automatiquement le nom du tableau en pointeur vers son premier élément, puis ajoute `i` éléments (en tenant compte de la taille du type) et déréférence le résultat. C'est ainsi que l'opérateur `[]` est implémenté en interne par le compilateur.

Question 17. Réponse : **C)**

`void*`

La fonction `malloc` retourne un `void*`, c'est-à-dire un pointeur générique qui peut pointer vers n'importe quel type. Il faut ensuite le convertir (cast) vers le type souhaité, par exemple `(int*)malloc(sizeof(int))`. Le type `void*` permet à `malloc` d'être une fonction universelle d'allocation, indépendante du type des données.

Question 18. Réponse : **B)**

Double libération (double free) — comportement indéfini

Appeler `delete` deux fois sur le même pointeur constitue une double libération (double free), ce qui provoque un comportement indéfini. La première fois, la mémoire est correctement libérée. La seconde fois, la zone mémoire a déjà été rendue au système et tenter de la libérer à nouveau corrompt les structures internes de l'allocateur. Pour éviter cela, on peut mettre le pointeur à `nullptr` après `delete`.

Question 19. Réponse : **A)**

`10 10`

L'instruction `*p = *q` copie la valeur pointée par `q` (soit `b = 10`) dans l'emplacement pointé par `p` (soit `a`). Après cette instruction, `a` vaut `10`. La variable `b` n'est pas modifiée. Attention : on copie ici la valeur, pas le pointeur. Si on avait écrit `p = q`, on aurait changé l'adresse stockée dans `p` sans modifier `a`.

Question 20. Réponse : **C)**

60

`p` pointe vers le début du tableau, soit `tab[0]`. L'expression `*(p + 1)` donne `tab[1] = 20`, et `*(p + 3)` donne `tab[3] = 40`. La somme est `20 + 40 = 60`. L'arithmétique des pointeurs avance par unités de la taille du type pointé : `p + 1` avance de `sizeof(int)` octets.

Question 21. Réponse : **C)**

Comportement indéfini : `a` est détruite à la fin de `createValue`, le pointeur retourné est dangling

La variable `a` est locale à la fonction `createValue` et stockée sur la pile. Elle est détruite à la fin de la fonction. Le pointeur retourné contient l'adresse d'une variable qui n'existe plus : c'est un pointeur dangling. Accéder à `*p` dans `main` est un comportement indéfini. La solution serait d'utiliser `new int(42)` pour allouer sur le tas, ou de retourner la valeur directement.

Question 22. Réponse : **C)**

Les lignes A et B

La déclaration `const int* const p` signifie que ni la valeur pointée ni le pointeur lui-même ne peuvent être modifiés. La ligne A (`*p = 20`) tente de modifier la valeur pointée, ce qui est interdit par le `const` à gauche. La ligne B (`p = nullptr`) tente de modifier l'adresse contenue dans `p`, ce qui est interdit par le `const` à droite. Les deux lignes provoquent donc des erreurs de compilation.

Question 23. Réponse : **B)**

`{1, 10, 3, 4, 5}`

Le pointeur `p` pointe vers `tab[2]` (valeur 3). L'expression `p[-1]` est équivalente à `*(p - 1)`, ce qui donne `tab[1]`. L'instruction `p[-1] = 10` modifie donc `tab[1]` de `2` à `10`. Le tableau résultant est `{1, 10, 3, 4, 5}`. L'arithmétique des pointeurs permet d'utiliser des indices négatifs pour accéder à des éléments situés avant la position courante du pointeur.

Question 24. Réponse : **C)**

Comportement indéfini : il faut utiliser `delete[]` pour libérer un tableau alloué avec `new[]`

Un tableau alloué avec `new[]` doit impérativement être libéré avec `delete[]`. Utiliser `delete` sans crochets sur un tableau est un comportement indéfini selon le standard C++. En pratique, cela peut provoquer une libération incomplète (seul le premier élément serait correctement détruit), une corruption de la mémoire, ou un crash. La règle est stricte : `new` avec `delete`, `new[]` avec `delete[]`.

Question 25. Réponse : **B)**

```
7 3
```

La fonction `swap` reçoit les adresses de `x` et `y` via des pointeurs. Elle échange les valeurs pointées en passant par une variable temporaire : `*a` (soit `x`) est sauvegardé dans `tmp`, puis `*a` reçoit `*b` (soit `y = 7`), puis `*b` reçoit `tmp` (soit `3`). Après l'appel, `x = 7` et `y = 3`. C'est l'idiome classique de passage par adresse pour modifier les variables de l'appelant.