QCM - Classes

Exercices - Chapitre 4 - Solutions [CSC-43043-EP] 2026

Solutions

Question 1. Réponse : **C)**

2

L'attribut `n` est initialisé à 0 par défaut (initialisation en place, C++11). Les deux appels à `increment()` incrémentent `n` à 1 puis 2. La méthode `valeur()` est marquée `const`, ce qui garantit qu'elle ne modifie pas l'objet, et retourne `n = 2`. Le programme affiche donc `2`.

Question 2. Réponse : **C)**

Le compilateur génère une erreur de compilation

En C++, les membres privés d'une classe ne sont accessibles que depuis les méthodes de la classe elle-même (et les fonctions amies). Toute tentative d'accès depuis l'extérieur de la classe provoque une erreur de compilation. Ce mécanisme d'encapsulation protège l'état interne de l'objet et garantit que seules les méthodes autorisées peuvent modifier les données.

Question 3. Réponse : **B)**

Il initialise l'objet sans nécessiter d'argument

Le constructeur par défaut est un constructeur qui ne prend aucun argument. Il est appelé automatiquement lorsqu'un objet est créé sans fournir de paramètres. Son rôle est de mettre l'objet dans un état valide et cohérent dès sa création, en initialisant les attributs avec des valeurs par défaut.

Question 4. Réponse : **C)**

```
constructeur() : x(val) {}
```

La liste d'initialisation s'écrit après le symbole `:` qui suit les parenthèses du constructeur, et avant le corps du constructeur entre accolades. Chaque attribut est initialisé avec la syntaxe `attribut(valeur)`. Cette syntaxe est préférable à l'affectation dans le corps du constructeur car elle initialise directement les membres sans passer par une étape intermédiaire de construction par défaut puis réaffectation.

Question 5. Réponse : **B)**

La méthode ne modifie pas l'état de l'objet

Le mot-clé `const` placé après la signature d'une méthode indique que cette méthode ne modifie pas l'état de l'objet. Le compilateur interdit toute modification des attributs non `mutable` à l'intérieur d'une telle méthode. De plus, seules les méthodes `const` peuvent être appelées sur un objet déclaré `const`.

Question 6. Réponse : **B)**

Il est appelé automatiquement quand l'objet est détruit pour libérer des ressources

Le destructeur est une fonction spéciale (préfixée par `~`) appelée automatiquement lorsque l'objet est détruit, c'est-à-dire à la fin de sa portée, lors d'un `delete`, ou quand le conteneur qui le possède est lui-même détruit. Son rôle principal est de libérer les ressources acquises par l'objet, comme la mémoire dynamique, les fichiers ouverts ou les ressources GPU.

Question 7. Réponse : **C)**

Un pointeur vers l'objet courant sur lequel la méthode est appelée

Le pointeur `this` est un pointeur implicite fourni par le compilateur dans chaque méthode non statique d'une classe. Il pointe vers l'objet courant sur lequel la méthode est appelée. Il est utile pour désambiguïser un paramètre portant le même nom qu'un attribut, pour retourner une référence sur l'objet courant (`return *this;`), ou pour passer l'objet courant à une autre fonction.

Question 8. Réponse : **C)**

`operator<<`

L'opérateur `<<` est surchargé avec `std::ostream&` comme premier paramètre pour permettre l'affichage d'un objet via `std::cout`. La signature typique est `std::ostream& operator<<(std::ostream& out, Type const& obj)`. Cet opérateur est défini comme fonction non-membre et retourne la référence sur le flux pour permettre le chaînage des affichages.

Question 9. Réponse : **B)**

Un mécanisme pour regrouper des symboles sous un préfixe commun afin d'éviter les conflits de noms

Un espace de noms (`namespace`) est un mécanisme du langage C++ qui permet de regrouper des fonctions, types et constantes sous un préfixe commun. Son objectif principal est d'éviter les conflits de noms entre différentes parties d'un projet ou entre bibliothèques. L'exemple le plus connu est `std`, qui contient tous les symboles de la bibliothèque standard.

Question 10. Réponse : **A)**

Ligne 1 uniquement

Dans une `class`, les membres sont privés par défaut. Ici, `x` est déclaré avant le mot-clé `public:`, il est donc privé. La ligne `a.x = 1` tente d'accéder à un membre privé depuis l'extérieur de la classe, ce qui provoque une erreur de compilation. En revanche, `y` est déclaré dans la section `public:`, donc `a.y = 2` est valide.

Question 11. Réponse : **B)**

Il empêche les conversions implicites via ce constructeur

Le mot-clé `explicit` empêche le compilateur d'utiliser un constructeur à un argument pour effectuer une conversion implicite. Par exemple, sans `explicit`, un constructeur `vec3(float v)` permettrait d'écrire `vec3 b = 1.0f;`, ce qui est une conversion implicite de `float` vers `vec3`. Avec `explicit`, seul l'appel direct `vec3 a(1.0f)` est autorisé, ce qui rend le code plus sûr et plus lisible.

Question 12. Réponse : **B)**

Parce que ces membres doivent être initialisés immédiatement et ne peuvent pas être assignés après construction

Les attributs `const` et les références doivent être initialisés au moment de leur création et ne peuvent pas être réassignés par la suite. La liste d'initialisation du constructeur est le seul endroit où cette initialisation peut avoir lieu. Dans le corps du constructeur, les membres sont déjà construits, et une affectation à un `const` ou une référence non initialisée est impossible, ce qui provoque une erreur de compilation.

Question 13. Réponse : **B)**

`a` vaut `{1, 2, 3}`

L'opérateur `+` est défini comme une fonction non-membre qui prend le premier argument `a` par valeur (copie). La modification via `a += b` affecte donc la copie locale, pas l'objet original. L'objet `a` dans `main` conserve ses valeurs initiales `{1, 2, 3}`. C'est un patron de conception courant : on implémente `+` en termes de `+=` en passant le premier argument par copie.

Question 14. Réponse : **B)**

`x` est protected dans `Circle` et accessible depuis ses méthodes

Les membres `protected` de la classe de base restent `protected` dans une classe dérivée héritant en mode `public`. Cela signifie qu'ils sont accessibles depuis les méthodes de la classe dérivée (et de ses propres dérivées), mais pas depuis l'extérieur. La méthode `get_x()` de `Circle` peut donc accéder à `x`, mais un code extérieur ne pourrait pas écrire `c.x` directement.

Question 15. Réponse : **B)**

La méthode est une fonction virtuelle pure et la classe est abstraite

La syntaxe `= 0` déclare une fonction virtuelle pure (*pure virtual*). Cela signifie que la classe ne fournit pas d'implémentation pour cette méthode et que toute classe dérivée concrète doit l'implémenter. Une classe contenant au moins une méthode virtuelle pure est abstraite : elle ne peut pas être instanciée directement. Ce mécanisme permet de définir une interface que les classes dérivées doivent respecter.

Question 16. Réponse : **B)**

Pour que le destructeur de la classe dérivée soit correctement appelé lors d'un `delete` via un pointeur de base

Lorsqu'un objet dérivé est détruit via un pointeur vers la classe de base (par exemple avec `delete ptr`), le compilateur appelle le destructeur correspondant au type statique du pointeur. Si le destructeur de la classe de base n'est pas virtuel, seule la partie base est détruite, et le destructeur de la classe dérivée n'est jamais appelé. Cela peut provoquer des fuites de ressources et un comportement indéfini. Un destructeur virtuel garantit l'appel correct du destructeur dérivé.

Question 17. Réponse : **B)**

La perte de la partie dérivée d'un objet lorsqu'il est copié dans un objet de type base

Le *slicing* se produit lorsqu'un objet de type dérivé est copié dans un objet de type base par valeur. Seuls les membres de la classe de base sont copiés, et toute la partie spécifique à la classe dérivée est perdue. C'est pourquoi le polymorphisme en C++ nécessite l'utilisation de pointeurs ou de références vers la classe de base, et non de copies d'objets.

Question 18. Réponse : **B)**

Il est unique et partagé par toutes les instances de la classe

Un attribut `static` dans une classe existe en un seul exemplaire, indépendamment du nombre d'objets créés. Il appartient à la classe elle-même et non à une instance particulière. Il peut être accédé via `NomClasse::attribut` sans avoir besoin d'un objet. Sa définition (avec initialisation) doit être faite une seule fois dans un fichier `.cpp`, en dehors de la déclaration de la classe.

Question 19. Réponse : **B)**

Parce que cela pollue l'espace de noms de tous les fichiers qui incluent cet en-tête

L'instruction `using namespace std;` dans un fichier `.hpp` importe tous les noms du namespace `std` dans l'espace de noms global. Comme un en-tête est inclus par de nombreux fichiers source, cette directive se propage à tous ces fichiers, créant potentiellement des conflits de noms involontaires. La bonne pratique est de n'utiliser `using namespace` que dans des fichiers `.cpp` locaux, et de préférer les qualificateurs explicites comme `std::vector` dans les en-têtes.

Question 20. Réponse : **B)**

La version `const`

L'objet `b` est déclaré `const`, donc le compilateur sélectionne automatiquement la version `const` de `operator[]`, qui retourne une `float const&`. Cette distinction entre version `const` et non `const` d'une même méthode est un mécanisme fondamental en C++ : le compilateur choisit la surcharge appropriée en fonction du caractère `const` de l'objet sur lequel la méthode est appelée.

Question 21. Réponse : **B)**

`0`

La fonction `print_area` prend un paramètre de type `Shape` par valeur, ce qui provoque un *slicing*. Lorsque l'objet `Circle` est copié dans le paramètre `Shape s`, la partie dérivée (incluant l'attribut `r` et la vtable de `Circle`) est perdue. L'appel `s.area()` utilise donc `Shape::area()`, qui retourne `0.0f`. Pour que le polymorphisme fonctionne, il faudrait passer le paramètre par référence ou par pointeur (`Shape const& s` ou `Shape const* s`).

Question 22. Réponse : **B)**

Erreur de compilation : `Rectangle` est abstraite car `area() const` n'est pas redéfinie

La méthode `Shape::area()` est déclarée `virtual float area() const = 0` (avec `const`). Dans `Rectangle`, la méthode `area()` est définie sans `const` : `float area()`. Ces deux signatures sont différentes. La méthode de `Rectangle` ne redéfinit donc pas la version pure virtuelle de `Shape`. Par conséquent, `Rectangle` hérite toujours de la méthode virtuelle pure `area() const` sans la redéfinir, ce qui rend `Rectangle` abstraite. L'instanciation `Rectangle r(3.0f, 4.0f)` provoque une erreur de compilation.

Question 23. Réponse : **A)**

`0`

Lors de la construction de `Derived d`, le constructeur de `Base` est appelé d'abord (`count` passe de 0 à 1), puis le constructeur de `Derived` (`count` passe de 1 à 2). Lors de la destruction (fin du bloc), le destructeur de `Derived` est appelé

d'abord (`count` passe de 2 à 1), puis le destructeur de `Base` (`count` passe de 1 à 0). Le programme affiche donc `0`. L'ordre de destruction est l'inverse de l'ordre de construction.

Question 24. Réponse : **B)**

Affiche `A` car `f()` n'est pas virtuelle, donc la résolution est statique

La méthode `f()` dans `A` n'est pas déclarée `virtual`. Par conséquent, l'appel `ptr->f()` est résolu statiquement selon le type du pointeur, qui est `A*`. C'est donc `A::f()` qui est appelée, affichant `A`. Pour obtenir un comportement polymorphique (appeler `B::f()`), il faudrait déclarer `f()` comme `virtual` dans la classe `A`. Sans `virtual`, la résolution se fait à la compilation et non à l'exécution.

Question 25. Réponse : **A)**

`n = 3`

Trois objets sont créés : `a` (id=0, next_id passe à 1), `b` (id=1, next_id passe à 2), `c` (id=2, next_id passe à 3). Après ces créations, `Object::get_next_id()` retourne 3 et `a.get_id()` retourne 0. Donc `n = 3 + 0 = 3`. L'attribut statique `next_id` est partagé par toutes les instances et sert de compteur global, tandis que `id` est un attribut d'instance propre à chaque objet.