Préparé par : Maxime DELAPORTE et Christopher DACCACHE

CRTP INFO31

Question 4 :

Phases de compilation :

1. Prétraitement : les fichiers .h sont inclus et les directives #define traitées.
2. Compilation : chaque fichier .cpp est traduit en code objet (.o) et vérifié pour les types et la syntaxe.
3. Édition des liens : les fichiers objets sont assemblés pour produire l'exécutable final, en résolvant les références entre fichiers.

Séparer déclarations (.h) et définitions (.cpp) permet de mieux organiser le code, de réutiliser les interfaces dans plusieurs fichiers et de compiler plus rapidement sans recompiler tout le projet à chaque modification.

Question 6 :

Il n'est pas nécessaire de nommer les paramètres dans la déclaration car le compilateur n'utilise que leurs types pour vérifier la signature. Les noms sont requis uniquement dans la définition pour pouvoir manipuler les valeurs. Les noms dans la déclaration sont facultatifs et servent juste à la lisibilité.

Question 7 :

Le mot-clé const après une méthode indique qu'elle ne modifie pas l'état de l'objet. Cela permet d'appeler la méthode sur des objets const et garantit la sécurité. Son absence signifie que la méthode peut modifier l'objet.

Question 9 :

Mettre l'opérateur << comme friend permet à cette fonction d'accéder aux membres privés de la classe pour afficher son contenu. Sans friend, l'opérateur ne pourrait pas lire directement les attributs internes de l'objet.

Question 13 :

C'est un passage par référence constante.

1. Le & évite de copier tout l'objet Grille en mémoire, ce qui est plus performant.
2. Le const garantit que la méthode choisir ne modifiera pas la grille. Elle doit seulement la consulter pour l'afficher et vérifier la validité du coup, elle ne doit pas la changer.

Question 17 :

Le compilateur ne vérifie que les déclarations. Il a seulement besoin de savoir que la méthode existe et que ses paramètres sont corrects pour valider la syntaxe du fichier.

C'est l'éditeur de liens, une étape qui arrive après la compilation, qui générera une erreur s'il ne trouve pas le code réel de la méthode pour construire l'exécutable final.

Question 22 :

La relation d'héritage s'exprime lors de la déclaration de la classe fille en utilisant deux points suivis du mode de visibilité et du nom de la classe mère.

Question 25 :

Création de m

Mere::Mere m

Explication : Création d'un objet Mere sur la pile. Le constructeur de Mere est appelé simplement.

Création de f

Mere::Mere f

Fille::Fille f

Explication : Création d'un objet Fille sur la pile.

1. La partie "Mère" de l'objet doit être construite en premier. Le constructeur de Mere est appelé.
2. Ensuite, le corps du constructeur de Fille est exécuté.

Création de p

Mere::Mere p

Explication : Allocation dynamique. Pointeur de type Mere* pointant vers un objet Mere.

Création de q

Mere::Mere q

Fille::Fille q

Explication : Allocation dynamique. Pointeur de type Fille* pointant vers un objet Fille. Comme pour f, on construit d'abord la partie mère, puis la fille.

Création de r

Mere::Mere r

Fille::Fille r

Explication : C'est le cas du polymorphisme. On a un pointeur de type parent (Mere*) qui pointe vers un objet enfant (Fille). La construction reste identique à un objet Fille normal (Mère puis Fille).

Création de z

Mere::Mere z

Fille::Fille z

Fille::~Fille z

Mere::~Mere z

Explication : C'est le cas du Slicing (troncature).

1. Fille { "z" } crée un objet temporaire de type Fille. Cela affiche "Mere::Mere z" puis "Fille::Fille z".
2. Cet objet temporaire est utilisé pour initialiser z (qui est de type Mere). Le compilateur utilise le constructeur de copie implicite de Mere. Ce constructeur copie la partie Mere de l'objet temporaire vers z, mais ignore la partie Fille. Ce constructeur de copie par défaut n'affiche rien.
3. L'instruction est finie, l'objet temporaire Fille est détruit immédiatement. Cela déclenche son destructeur : "Fille::~Fille z" puis "Mere::~Mere z".
4. À la fin, il reste un objet z de type Mere sur la pile

Partie de foo!

Mere::foo m

Explication : m.foo(); -> Appel standard sur un objet Mere.

Fille::foo f

Explication : f.foo(); -> Appel standard sur un objet Fille. La méthode est surchargée (override).

Mere::foo p

Explication : p->foo(); -> p est un Mere* pointant sur un Mere.

Fille::foo q

Explication : q->foo(); -> q est un Fille* pointant sur une Fille.

Fille::foo r

Explication : r est un pointeur Mere*, mais il pointe réellement vers une Fille. Comme foo est déclarée virtual, le programme regarde la "table des méthodes virtuelles" (vtable) au moment de l'exécution et appelle la version de l'objet réel (Fille), pas celle du pointeur.

Mere::foo z

Explication : L'objet z a subi une troncature (slicing) lors de sa création. Bien qu'initialisé avec une Fille, z est purement un objet de type Mere. Il n'a plus aucune information sur Fille. Il appelle donc Mere::foo.

Partie de bar!

Mere::bar m

Explication : Appel classique.

Fille::bar f

Explication : Appel classique. Fille a sa propre méthode bar qui masque celle de Mere.

Mere::bar p

Explication : Appel classique via pointeur.

Fille::bar q

Explication : Appel classique via pointeur Fille*.

Mere::bar r

Explication : La méthode bar n'est PAS déclarée virtual dans Mere. Le compilateur fait donc une "liaison statique" (static binding). Il regarde uniquement le type du pointeur (Mere*). Il appelle donc Mere::bar, même si l'objet pointé est une Fille.

Mere::foo z

Explication : Même explication que précédemment, z est une Mere.

Fin de partie!

Fille::~Fille r

Mere::~Mere r

Explication : Explication : r est un pointeur Mere*. Cependant, le destructeur de Mere est déclaré virtual. C'est crucial. Cela permet au programme d'appeler le destructeur de l'objet réel (Fille) d'abord.

1. ~Fille() est appelé ("Fille::~Fille r").
2. Automatiquement, le destructeur parent ~Mere() est appelé ensuite ("Mere::~Mere r").

Fille::~Fille q

Mere::~Mere q

Explication : delete q; -> Destruction standard d'un objet Fille.

Mere::~Mere p

Explication : delete p; -> Destruction standard d'un objet Mere.

Mere::~Mere z

Explication : Destruction de z. Rappelez-vous que z est de type Mere. Seul le destructeur de Mere est appelé.

Fille::~Fille f

Mere::~Mere f

Explication : Destruction de f. D'abord le destructeur de la classe fille, puis celui de la mère.

Mere::~Mere m

Explication : Destruction de m.