Université de Reims Champagne-Ardenne
UFR Sciences Exactes et Naturelles
Licence Informatique
INFO0402 : Programmation Orientée Objet

Devoir sur table
du 17 mars 2017

Aucun document n'est autorisé.

Questions de cours

1. Donner tous les usages possibles de `const`.
2. Donner l'ensemble des cas concrets où un destructeur est appelé.
3. Expliquer la différence entre la notion de lien anticipé et de lien tardif.
4. Quelle est la différence entre une classe de base virtuelle pure et une interface virtuelle pure ?
5. Quelle est la valeur de la fonction sizeof(T) si le type `T` est défini de la manière suivante :
```
class T {
    static int  v;
    const  char c[2];
    double      *pt;
};
```
Si la réponse dépend de paramètres non précisés dans la question, on indiquera lesquels et on donnera la valeur de `sizeof(T)` dans le contexte choisi.
6. Qu'est-ce-que l'upcasting ? On donnera un exemple concret.
7. Pourquoi, lorsqu'un objet peut être polymorphique, son destructeur doit absolument être déclaré comme virtuel.
8. Pourquoi dit-on qu'une architecture doit être à forte cohésion et à faible couplage ?
9. Quelle est la différence entre une agrégation et une composition ?
10. Comment cette différence se traduit-elle lorsque l'on implémente une classe en `C++` ?

Exercice 1 : approche objet

On veut créer un ensemble de classes permettant de stocker des objets constitués d'animaux (singe, chat, lapin), et de leurs caractéristiques : s'il est herbivore ou carnivore, familier (avec un nom) ou sauvage, son habitat.

- les objets devront être polymorphes : on pourra regarder chacun des objets sous différentes formes (comme animal ou carnivore),
- un singe est la fois herbivore et carnivore.
- on peut construire un chat sauvage ou un chat familier (à savoir toute combinaison d'animaux familiers ou sauvages),
- un animal peut crier (méthode virtuelle).
- un animal peut manger (méthode virtuelle), son implémentation concrète dépendra du régime alimentaire.

1. On voudrait dans un premier temps adopter une approche purement hiérarchique. Donner la représentation UML associée.
2. Quel est l'inconvénient de l'approche précédente ?
3. On voudrait maintenant adopter une approche utilisant l'héritage multiple. Donner la représentation UML associée.
4. On ajoute la contrainte suivante : une fois créé, un animal sauvage pourra devenir familier ou inversement sans être obligé de changer son type. Proposer une méthode afin de rendre cette contrainte possible.
5. Donner alors la représentation UML associée.
6. Écrire alors le code associé à l'organisation de classes obtenues à la question précédente. L'implémentation concrète des cris d'animaux sera faite sous forme d'un `cout` du cri associé à l'animal. L'implémentation concrète du regime alimentaire d'un animal sera faite sous forme d'un `cout` d'un aliment qu'il mange.
   (a) Donner la définition de la classe `animal`.
   (b) Donner la définition des classes `herbivore` et `carnivore`.
   (c) Donner la définition des classes `sauvage` et `familier`.
   (d) Donner la définition de la classe `singe`.
   (e) Donner la définition de la classe `chat`.
   (f) Donner la définition de la classe `lapin`.

Exercice 2 : commande

Afin de créer un gestionnaire de tâches, on veut créer une liste de tâches avec les règles suivantes :

- une tâche abstraite `Task` dispose des méthodes `start()` (pour lancer la tâche), `is_complete()` (qui retourne vrai si la tâche est terminée), `is_running()` (qui retourne vrai si la tâche est en cours d'exécution), `completed()` (pour signaler au gestionnaire que la tâche est terminée).
- le stockage sous-jacent de la liste de tâche est une liste doublement chainée (dont les cellules sont nommées `TaskCell`),
- l'insertion dans cette liste s'effectue toujours à la fin.
- les tâches situées au début de la liste sont considérées comme prioritaire.

Cette liste de tâches est gérée dans un `TaskManager` dont le but est de toujours faire un sorte qu'il n'y ait jamais plus de n tâches actives en même temps. Les opérations de gestion de la liste (insertion en fin, suppression d'un élément) seront déléguées au `TaskManager`.

On pose alors les questions suivantes (il est conseillé de les lire en entier avant de commencer l'exercice) :

1. Expliquer comment procéder afin qu'une `Task` puisse signaler sa terminaison au `TaskManager`.
2. Dessiner le diagramme UML associé aux classes `TaskManager`, `Task`, `TaskCell`, et `ConcreteTask` où `ConcreteTask` est l'implémentation d'une tâche concrète. Attention de bien penser aux différents liens, en particulier à la façon de réaliser pour une `Task` le signalement de sa terminaison à `TaskManager`.
3. On veut maintenant donner l'implémentation de la classe `Task` permettant de représenter tout type de tâche pouvant se conformer aux méthodes de `Task`.
   (a) Donner la déclaration de la classe abstraite `Task` (aucun détail de méthodes ne devra être donnée à cette question).

(b) Donner le constructeur de `Task`.
(c) Donner le destructeur de `Task`.
(d) Est-il possible de donner une implémentation par défaut de `start()` ? Si oui, on la donnera ; si non, on expliquera pourquoi.
(e) Même question pour `is_complete()`.
(f) Même question pour `is_running()`.
(g) Même question pour `complete()`.

4. On veut maintenant donner l'implémentation d'un tâche concrète `ConcreteTaskA` (concrete = aucune méthode ne doit rester virtuelle). Cette tâche sera définie à partir d'une autre classe `A` qui a un constructeur par défaut et qui exécute une tâche lorsque l'on lance sa méthode `execute()` ; la tâche étant terminée au moment où cette méthode a fini de s'exécuter.
   (a) Donner la déclaration de la classe `ConcreteTaskA`.
   (b) Donner le constructeur de `ConcreteTaskA`.
   (c) Donner le destructeur de `ConcreteTaskA`.
   (d) S'il n'existe pas d'implémentation par défaut suffisante de `start()`, donner sa spécialisation pour `ConcreteTaskA`.
   (e) Même question pour `is_complete()`.
   (f) Même question pour `is_running()`.
   (g) Même question pour `complete()`.
5. On s'intéresse maintenant à l'implémentation d'un `TaskCell`, qui permet de chainer un ensemble de `Task`s abstraites.
   (a) Expliquer votre choix de manière à faire en sorte que `TaskManager` puisse accéder à la totalité des champs de `TaskCell` sans restriction.
   (b) Expliquer pourquoi `TaskCell` ne peut faire référence à une tâche quelconque qu'à travers un pointeur sur `Task` ?
   (c) Donner la déclaration de la classe `TaskCell`.
   (d) Donner le code du constructeur de `TaskCell`.
   (e) Donner le code du destructeur de `TaskCell`.
6. On s'intéresse maintenant à l'implémentation du `TaskManager`. On rappelle que le `TaskManager` devra fait en sorte qu'il n'y ait jamais plus de n tâches en cours d'exécution en même temps (où n est un paramètre de la classe). Aucune tâche ne devra être lancée tant que le TaskManager n'est pas explicitement démarré (voir `Start()`) ou s'il est arrêté (voir `Stop()`).
   Enfin, on supposera que l'appel à la méthode `Task::start()` revient immédiatement (*i.e.* la tâche lancée est en cours d'exécution en parallèle ; `Task::is_running()` est vrai et n'est terminée que lorsque `Task::is_complete()` est vrai). Tous les problèmes potentiels liés à une exécution parallèle seront ignorés.
   (a) Donner la déclaration de la classe `TaskManager`.
   (b) Donner le constructeur de `TaskManager`.
   (c) Donner le destructeur de `TaskManager`.
   (d) Donner le code de la méthode `Start()` qui permet de démarrer le `TaskManager`.
   (e) Donner le code de la méthode `Stop()` qui permet d'arrêter le `TaskManager`. On laissera les tâches en cours d'exécution s'arrêter d'elles-mêmes.
   (f) Donner le code de la méthode `void AddTask(Task *)` qui permet d'ajouter une tâche à la liste des tâches. Si le condition nécessaires sont réunies, la tâche devra démarrer immédiatement.

(g) Donner le code de la méthode privée `Signal` (dont vous devez choisir les arguments) appelée par une tâche qui vient de se terminer. Cette tâche devra alors être supprimée de la liste des tâches, et si les conditions sont réunies, une nouvelle tâche de remplacement sera exécutée.

Exercice 3 : allocateur

On rappelle que :

- `::operator new(n)` effectue une allocation mémoire non typée de `n` bytes.
- `::operator delete(mem)` libère une mémoire allouée non typée.
- `new(adr) A(...)` lance le constructeur en place `A(...)` à l'adresse mémoire `adr` (déjà allouée).
- `a-> A()` lance le destructeur en place d'un objet `A` placé à l'adresse `a`.

On voudrait écrire un allocateur (classe `Allocator`) dont le rôle est d'allouer un bloc mémoire capable de stocker `n` éléments de type `A`, puis de distribuer la mémoire qu'il a réservé lorsqu'on lui en fait la demande. Une fois l'allocateur construit,

- la méthode `A* Contruct(...)` choisit un emplacement mémoire non occupé dans le bloc, et y lance le constructeur en place `A(...)` (où `...` représente juste les arguments du constructeur). Il y aura donc autant de méthodes Construct que de constructeurs différents.
- la méthode `void Destruct(A* ptr)` lance le destructeur sur l'élément du bloc pointé par `ptr`, et remet `ptr` dans le pool des emplacements mémoires libres du bloc mémoire.

On suppose que le type `A` possède un constructeur par défaut `A()`, un constructeur prenant en paramètre un entier `A(int)`, et un constructeur par copie `A(const A&)`. Il est conseillé de lire l'ensemble des questions avant de commencer l'exercice afin de comprendre la logique de la classe.

1. Afin de gérer le bloc de mémoire, l'allocateur doit également posséder une table d'allocation. Expliquer quelle est la nature de cette table, de manière à ce qu'elle permette de savoir quelles sont les places libres (Construct = retirer un élément de cette liste, Destruct = ajouter un élément de cette liste). Une allocation ne devra générer aucune recherche dans cette table.
2. Écrire le constructeur `Allocator(n)`. L'allocateur résultant allouera un bloc assez grand pour stocker `n` éléments de type `A` sans les construire. Il possèdera aussi la table d'allocation décrite à la question précédente.
3. Écrire le destructeur `~Allocator()`. S'il reste des éléments construits, ils devront être détruits avant de désallouer le bloc mémoire de l'allocateur.
4. Écrire la méthode `Construct(...)` qui trouve une place libre dans la table d'allocation, la prend, lance le constructeur `A(...)` sur cet emplacement, et retourne un pointeur vers l'élément construit.
5. Écrire la méthode `Destruct(ptr)` qui détruit l'élément pointé par `ptr`, et marque l'élément comme libre. Remarquer que si `mem` est un pointeur vers le début du bloc alloué, alors avec l'algèbre des pointeurs, `ptr-mem` donne le numéro du bloc libéré.
6. Donner un exemple concret d'utilisation de l'allocateur ci-dessus.
7. Expliquer pourquoi l'utilisation d'un allocateur est généralement une bonne idée lorsque l'on utilise des conteneurs de type liste chainée ou arbre ?