POO et Maranzana - 3IF

Victor Lezaud

19 novembre 2017

Sommaire

1 Programmation Orienté Objet

1.1 Classes et Objets

1.1.1 Qu'est-ce qu'un objet ?

En programmation on peut considérer un objet comme une forme de boîte noire avec des boutons sur l'extérieur. On ne sait rien sur ce que l'y a à l'intérieur mais on peut l'utiliser grâce aux commandes qui sont à l'extérieur. Un objet possède des données, nommées attributs qui définissent ce que c'est (couleur d'un véhicule) et des méthodes qui définissent ses comportements (comment ce véhicule se déplace).

1.1.2 Qu'est-ce qu'une classe ?

La classe est le plan qui permet de construire la boite noire qu'est l'objet, elle définit sa composition, son fonctionnement et son interface extérieure. On crée donc un objet à partir d'une classe, on dit aussi qu'un objet est une instance d'une classe. C'est la classe qui définit les attributs et méthodes de l'objet. En programmation une classe est un type.

1.2 Encapsulation des données

1.2.1 Qu'est-ce ?

Le principe d'encapsulation des données oblige le programmeur à limiter l'accès aux données présentes dans les objets. Il faut donc essayer au maximum que les données d'un objet ne puissent être modifiés que par l'objet lui-même. C'est de la que vient l'aspect boite noire de la description ci-dessus. Le but de l'informatique étant le traitement automatisé de données il est évident que l'on va vouloir modifier les données d'un objet. Mais cela ne doit se faire qu'au travers l'appel de méthodes qui sont don les commandes extérieures dans notre image.

1.2.2 Pourquoi ?

Le principe d'encapsulation permet de sécuriser les données et donc le fonctionnement d'un objet. En effet cela permet au programmeur de maîtriser où sont modifiées les données et comment elles le sont. Cela permet ensuite à un programmeur d'assurer le bon fonctionnement de sa classe et donc de la partager avec d'autres en ne leur donnant que la description des méthodes, de ce qu'elles font uniquement.

1.3 Héritage

1.3.1 Principe

En POO, les classes permettent d'implémenter dans le code des concepts tels qu'un ensemble ou un animal. Dans une application nous allons souvent utiliser des concepts très proches les uns des autres, par exemple dans un jeu vidéo le personnage du joueur est très proche de ceux contrôlés par l'IA. Ils vont donc partager une bonne partis de leur données et de leurs comportements, vu

qu'ils sont tous deux une sorte de personnage.Nous allons donc devoir réécrire le même code à plein d'endroits différents et non seulement ce sera une perte de temps mais en plus la modification sera extrêmement pénible. La solution est de définir une classe mère personnage qui contiendra le comportement et les données communs aux deux classes filles.

1.3.2 Concrètement

Lorsque l'on codera on créera une relation d'héritage chaque fois qu'on pourra dire que X est une sorte de Y. Y sera la classe dite mère et X la classe dite fille. X héritera donc de tout ce qui caractérise Y. L'héritage impose une règle majeure au programmeur : tout ce qui est fait avec Y doit pouvoir être fait avec X. Cela paraît simple mais peut vite devenir très compliqué !

1.3.3 Qui peut contenir qui

La règle : Dans une relation d'héritage : X est une sorte de Y. On peut stocker un objet X dans une variable de type Y, mais on ne peut stocker un objet Y dans une variable X.

Pourquoi : Comme on l'a dit dans l'héritage X possède toutes les caractéristiques de Y et tout ce qui est fait avec Y peut-être fait avec X. Stocker un objet X dans une variable Y ne pose donc aucun problème. Par contre X peut avoir des méthodes qu'Y n'a pas, on ne peut donc pas stocker d'objet Y dans une variable X car l'appel de certaines méthodes de X ne pourrait être supportés par l'objet Y.

1.4 Polymorphisme

1.4.1 Principe

Grâce au polymorphisme, des éléments d'un type général identique peuvent se comporter différemment en fonction de leur implémentation. Autrement dit, une même opération peut se comporter différemment pour différentes classes issues d'une même arborescence d'héritage

1.4.2 En plus simple

Encore une fois on reprends l'exemple X est une sorte de Y. Imaginons que l'on appelle une méthode Afficher de Y à partir d'une variable de ce type. Si la variable contient un objet issu de Y, aucun problème. Si la vairable contient un objet issu de X :

- Sans polymorphisme : l'objet s'affiche comme un objet issu de Y et on perde de l'information.
- Avec polymorphisme : l'objet se comporte bien comme un objet issu de X et on ne perd aucune information.

1.4.3 Utilité

Le polymorphisme est l'un des outils les plus utilisés en POO. Imaginons que nous voulion réaliser un moteur 3D. Il faudra effectuer un rendu de l'ensemble des objets présent dans la scène mais chaque objet va avoir son propre

comportement et donc sa propre méthode de rendu. Il est impensable d'avoir une variable pour chaque objet présent sur la scène, ou même d'avoir une collection pour chaque type d'objet présent. On va donc se servir de l'héritage et du polymorphisme en définissant un type Objet3D dont tous les autres vont hérités et on va appelé la méthode rendu pour chaque élément de la collection d'Objet3D. Le bon fonctionnement est assuré par le polymorphisme.

1.5 Statique / Dynamique

1.5.1 Allocation dynamique

Une allocation est dynamique lorsque l'on ne peut pas déterminer la taille de l'espace qui doit être alloué avant l'exécution du programme. Dans ce cas le compilateur ne peut pas réaliser d'allocation statique. Cela arrive pour des tableaux dont la taille est définie par une variable ou la construction d'objets.
Dans la cas d'une allocation dynamique le compilateur ne peut plus gérer la libération de l'espace mémoire de façon statique, il revient donc au programmeur d'expliciter quand il faut libérer la mémoire.
On utilise l'opérateur `new` pour l'allocation dynamique et l'opérateur `delete` pour libérer la mémoire.

1.5.2 Liaison dynamique

Quand on ne peut définir avant l'exécution quelle est la méthode qui sera appelé (à cause du polymorphisme) le compilateur établit un lien non plus statique mais dynamique.
En C++ les liens dynamiques ne peuvent exister que dans le cadre de méthodes déclarées `virtual`.

2 Vocabulaire

Quelques précisions sur certains détails de vocabulaire chers à Maranzana

2.1 Fonctions

2.1.1 Fonction et Procédure

Une fonction telle que celle du C est dite "fonction ordinaire exportée" ou "fonction ordinaire non exportée" si elle est `static`.
Une fonction déclarée dans une classe est nommée "méthode" ou "méthode de classe si elle est `static`. Toute fonction de type `void` et nommée "procédure".

2.1.2 Paramètre formels et effectifs

Paramètre formel : est défini à la déclaration de la fonction. Le paramètre formel n'est lié qu'à la fonction et aucunement à son utilisation.

Paramètre effectif : est la valeur donnée lors de l'appel de la fonction. Chaque paramètre effectif correspond à un paramètre formel de la fonction appelée.

2.2 Pointeurs

Soit ptr un pointeur et tab un tableau :
— *ptr est "L'accès à la valeur pointée par ptr"
— tab[i] est "L'accès à la $i^{ème}$ valeur après la valeur pointée par tab"

3 Mémoire

3.1 La pile

Les cases mémoires de la pile correspondent aux cases alloués de façon statique. C'est le cas le plus général. Toutes les variables qui sont déclarés ont une partie de la mémoire qui leur est alloué, cette partie de la mémoire est automatiquement libéré quand on sort du bloc de code où la variable est déclarée. D'où la notion de pile car la case libérée est toujours la dernière à avoir été allouée.

3.2 Le tas

Le tas contient les cases mémoires allouées dynamiquement soit avec l'opérateur `new`. On y retrouve donc des tableaux ou des objets reliées à un pointeur. Ces cases ne sont jamais libérées par le compilateur, il revient donc au programmeur de libérer lui-même les cases allouées dans le tas avec l'opérateur `delete`. Tout opérateur `new` doit avoir un `delete` associé et inversement, afin que toute la mémoire soit libérée.

3.3 Les constantes

Il existe une troisième "zone" de la mémoire, gérée intégralement par le langage, qui contient l'ensemble des constantes du programme.

4 Compilation

4.1 Déroulement de la compilation

Le passage du C++ à l'exécutable comporte deux grandes étapes la compilation et l'édition des liens

4.1.1 Compilation

La compilation transforme le code C++ en binaire translatable. La compilation transforme un fichier .cpp en fichier .o. Il faut une compilation pour chaque fichier .cpp. Les fichiers .h n'ont aucune influence sur le nombre de compilation.

4.1.2 Edition des liens

L'édition des liens est l'étape qui crée l'exécutable à partir des binaires translatables. Elle fait le lien entre les fonctions utilisés dans d'autres fichiers que celui où elles sont définies. Il y a toujours une et une seule édition des liens !!

4.2 Make et makefile

Le fonctionnement de la commande make est définit par le fichier makefile, elle permet de créer une version à jour de l'exécutable.
Voici un exemple de makefile usant de variables et de pattern.

```
#Modele de fichier makefile

RM=rm
ECHO=echo
COMP=g++
EDL=g++
RMFLAGS=-f
ECHOFLAGS=
COMPFLAGS=
EDLFLAGS=
INT=
REAL=$(INT:.h=.cpp)
OBJ=$(REAL:.cpp=.o)
EXE=a.out
CLEAN=efface

.PHONY:$(CLEAN)

$(EXE): $(OBJ)
$(ECHO) $(ECHOFLAGS) "EDL de demo"
$(EDL) -o $(EXE) $(OBJ)

%.o: %.cpp %.h
$(ECHO) $(ECHOFLAGS) "compil de <$<>"
$(COMPIL) $(COMPFLAGS) -c $<

$(CLEAN):
$(RM) $(RMFLAGS) $(EXE) $(OBJ)
```

5 Guide de style

5.1 Présentation

— Pour des problèmes d'impression et de facilité de lecture dans une fenêtre, limiter la longueur des lignes à 80 caractères.
— Ne pas s'acharner à écrire du code très court (illisible), mais ne pas le décomposer sans fin non plus et ne pas introduire de code inutile.
— entre code élaboré et rustique, choisir la compréhensibilité et la simplicité.
— Placer les # des directives du préprocesseur en première colonne afin qu'ils soient nettement visibles (pas d'indentation).
— Déclarer une seule variable par ligne.
— Ne truffez pas vos expressions de parenthèses inutiles qui finissent par polluer et le rendre illisible
— Dans les conditions et les boucles toujours placer les accolades même si elles ne sont pas nécessaires.
— N'écrire qu'une seule instruction par ligne
— Mettre les imbrications des instructions en valeur par indentation et aligner les ouvertures et fermetures de blocs
— Signaler les instructions `for` qui ne comportent pas de bloc.

5.2 Commentaire

— Expliquer les passages de l'algorithme au code ne présentant pas un caractère évident. Un commentaire doit expliquer et non dupliquer.
— La mise en page doit faciliter la lecture du code et des commentaires associés. Les commentaires noyés dans le code sont de moindre intérêt.
— Lorsque le #endif (ou le #else) est éloigné du #if, préciser la correspondance par un commentaire.

5.3 Mise au point et test

— Tester séparément chaque partie du programme en réalisant si besoin un programme de test dédié
— Essayer d'éviter les cas particuliers qui nécessitent des jeux de tests plus long

5.4 Constantes et macro-définition

— Pour définir les constantes, préférer au préprocesseur (#define) les formes const et enum dont la portée est plus facile à contrôler.
— Ne pas utiliser les macros-définitions et les remplacer par des fonctions.
— Définir les consternantes ayant un rapport entre elles dans un enum
— Limiter la portée des constantes au nécessaire
— Définir les constantes par des littéraux

5.5 Nom

— Toutes les variables doivent porter un nom rappelant leur signification. Il n'est pas utile que le nom rappelle le type.

— Ne pas donner à un objet le même nom qu'un autre objet du même espace de nommage.
— Ne pas commencer de nom par un blanc souligné ("_")
— Dans les noms composés de plusieurs mots, mettre une majuscule en tête de chaque mot.
— Ecrire les noms de constantes et les valeurs de type énumérés en majuscules
— Commencer les noms des objets publics par une majuscule et les autres par une minuscule
— Réserver les variables d'une seule lettre pour de simples indices de boucles ou des variables temporaires.

5.6 Expression

— Proscrire les expression trop complexes.
— Ne pas écrire d'expressions dont le résultat dépend de l'ordre dans lequel elle sera évaluée (`rang[i++]=i;`)
— Exploiter l'ordre d'évaluation des expressions pour éviter les évaluations inutiles ou interdites
— Calculer la taille des zones allouées avec sizeof

5.7 Formes algorithmiques

— Pour une boucle infinie, ne pas écrire `while(true)` mais plutôt `for(;;;)`
— Placer un `break` à la fin de chaque `case`. Commenter une absence volontaire
— Dans le cas d'une sélection portant sur une variable de type énuméré, traiter explicitement tous les cas.

5.8 Classe

— Distinguer dans chaque classe la partie interface (.h) qui est exportée de la partie réalisation (.cpp) qui est privée
— Construire de petites classes bien ciblées et à la sémantique cohérente plutôt qu'une grande classe qui fait tout.
— Chaque constructeur doit initialiser tous les attributs de l'objet.
— Le constructeur doit allouer lui-même tous les attributs de l'objet.
— Une classe dont un attribut est alloué dynamiquement dans le constructeur doit avoir un constructeur de copie et doit définir l'opérateur d'affectation.
— Le destructeur doit libérer toutes les zones allouées dans le constructeur.
— La destruction de tous les objets créés est impérative pour que la place mémoire soit récupérée. Pour s'en assurer, on pourra compter les créations et les destructions de toutes les instances de la classe et ses descendantes.
— Ne pas déclarer d'attribut public.
— Ne pas initialiser d'attribut par l'intermédiaire d'un autre attribut
— Ne pas fournir systématiquement les mutateurs (set()), cela viole le principe d'encapsulation

— Définir un constructeur de copie erroné lorsque son utilisation est illicite (déclaration sans définition). De même pour l'opérateur d'affectation
— Pour l'opérateur d'affectation attention aux écriture `x=x;`.
— L'en-tête de chaque méthode publique contient une description de ce que fait la méthode. Ne pas dire comment elle le fait

5.9 Héritage

— Il faut toujours déclarer un destructeur `virtual`.
— Pour avoir une classe abstraite sans méthode virtuelle pure il faut déclarer les constructeurs en `protected`.

5.10 Tableaux

— Pour comparer deux chaines il faut comparer les contenus et non les adresses.
— Utiliser `delete []` pour les tableaux