IFT1166 – TRAVAIL PRATIQUE #3 – 18 juin 2008

Gestion de stocks à la sauce C++!

Mohamed Lokbani

Équipes : le travail peut-être fait en binôme mais vous ne remettez qu'un travail par équipe.
Remise : une seule remise est à effectuer par voie électronique **le mercredi 09 juillet 2008, 12h00 au plus tard, sans possibilités de prorogation**.
Conseils : n'attendez pas le dernier jour avant la remise pour engager votre travail. Vous n'aurez pas le temps nécessaire pour le réaliser.

But : dans ce TP, vous allez réaliser un programme simple de gestion destiné à un magasin de vente et d'achat de disques d'occasion. Les connaissances développées dans ce TP sont essentiellement la manipulation de tableaux et la programmation orientée objet à l'aide de classes.

Énoncé : pour réaliser la gestion d'un magasin d'achat/vente de disques compacts (CD) vous devrez gérer deux "bases de données": le stock qui sera représenté par une table de tous les articles déjà vendus ou en instance de vente dans le magasin et les opérations qui seront représentées par une table de tous les articles vendus ou achetés.

Un article dans la base du stock contient au moins les informations suivantes :

titre: chaîne de caractères identifiant le CD.
auteur: nom de l'auteur principal (chaîne de caractères).
prix de vente: prix auquel est vendu le CD.
quantité: nombre d'exemplaires de ce CD en stock.
quantité max: nombre maximum de CD que le gérant du magasin souhaite avoir en stock. Le magasin n'achètera pas un CD d'un client si le nombre d'exemplaires de ce CD est déjà égal à cette quantité.

Toute opération (vente ou achat de CD) sera consignée dans la base des opérations. Les informations rattachées à une transaction sont au moins les suivantes :

type de la transaction: indique s'il s'agit d'une vente ou d'un achat (du point de vue du magasin)
montant de la transaction: de la vente ou de l'achat
indice dans la table des stocks: indice dans la table des stocks de l'article qui a été vendu ou acheté. Ceci permet de retrouver toutes les informations reliées à cet article.

Vous devez programmer une classe **Magasin** qui offre les méthodes suivantes :

Magasin(int capital) est le constructeur de la classe Magasin. Elle prend en argument le capital de départ (en dollars $CA) que le gérant du magasin dépose sur le solde du compte et à l'ouverture du magasin.

bool achat(String titre, String auteur) retourne vrai si l'achat est réalisable et faux dans le cas contraire. L'élément « titre » identifie de manière unique un CD. Un achat est possible si la quantité maximale en stock de ce CD n'est pas atteinte et si le solde du magasin est positif et supérieur au montant de l'achat. Le prix d'achat du CD contracté par le magasin est équivalent à 60% du prix de vente. Dans le cas où ce CD n'était pas déjà en stock, il convient de créer dans la base de stock, une fiche pour ce nouveau CD. Son prix de vente est par défaut fixé à 10$CA (le magasin l'achètera donc 6$CA) et sa quantité plafonnée sera fixée par défaut à 4.

Il appartient à cette méthode de mettre à jour les bases du stock et d'opérations de manière rationnelle.

bool vente(String titre) retourne vrai si la vente du CD identifié par son titre est possible. Une vente est possible dès lors que le CD est en stock. Lorsque la vente est réalisée, les informations sur le stock et les transactions sont mises à jour.

bool cdEnStock(String titre) retourne vrai si le CD dont le titre est spécifié en argument est disponible dans la base et qu'un exemplaire au moins est en stock.

int auteurEnStock(String auteur) retourne le nombre de CD différents en stock dont l'auteur principal est spécifié en argument.

int getSolde() retourne le solde (en $CA) du magasin.

int getNombreCDEnStock() retourne le nombre de CD disponibles dans le magasin.

void afficheCDLePlusVendu() affiche les renseignements relatifs au CD le plus commercialisé par le magasin.

void setQuantiteMax(String titre, integer quantite) permet de changer la quantité maximale du CD dont le titre est spécifié en argument par la valeur quantité. Cette méthode sera par exemple appelée par le gérant du magasin et ceux, afin de réduire le nombre maximal d'exemplaires d'un CD qui se commerciale difficilement ou au contraire augmenter ce nombre dans le cas d'un CD qui se vend aisément.

void afficheTransactions() méthode qui affiche les informations de toutes les transactions réalisées depuis la création du magasin. Les informations concernant une transaction sont : le montant de la transaction, le type de la transaction (achat ou vente), le titre du CD vendu ainsi que son auteur.

Notes importantes : seule l'écriture de la classe Magasin vous est demandée. Vous pouvez pour cela développer vos propres classes « satellites » si vous le souhaitez.

Afin de vous aiguiller, nous vous fournissons le code pour tester votre programme et une sortie écran produite par l'exécution de notre programme.

Hypothèses et contraintes :

- un des objectifs de ce travail est de séparer la partie classe de la partie code du test. Ceci écrit, comme première étape, vous pouvez mettre tout le code dans un seul fichier. Si le programme fonctionne correctement, vous pouvez passer à la seconde étape qui consiste à créer les fichiers « magasin.h » et « magasin.cpp ».
- pour commencer, vous devez déjà respecter les différentes contraintes de programmation précédemment décrites.
- IFT1166 est un cours en C++, donc votre programme doit être écrit en C++ et non pas en C ni en Java! Nous n'autoriserons aucune référence au langage C. Par exemple, l'instruction [#include <stdio.h>] fait référence au langage C donc elle n'est pas autorisée. Utilisez plutôt [#include <iostream>].
- ce travail n'est pas un exercice d'algorithmique donc pas besoin de compliquer le travail pour rien!
- vous devez vous assurer de ne pas changer les noms des fichiers et de respecter par la même occasion le format de l'affichage en sortie.

Rapport : le fichier « rapport.pdf » va décrire comment vous avez procédé pour réaliser ce travail (pour le contenu d'un rapport voir aussi la « FAQ » sur la page web du cours). Le rapport doit être au format « pdf » (voir là aussi la FAQ sur la page web du cours sur la façon de générer un fichier au format « pdf »).

Remise : il est important de noter que votre TP sera compilé avec gcc3.4.2. Si par choix, vous décidez d'utiliser un autre compilateur, vérifiez que le code que vous avez produit (devant normalement fonctionner correctement chez vous) fonctionne aussi sur les ordinateurs de la DESI. Pour avoir la version du compilateur, utilisez la commande "**gcc –v**", qui devra donner le numéro de version "**3.4.2**".
Par ailleurs, assurez-vous de la présence de l'option « pedantic » sur la ligne de compilation. (Cette option n'est pas activée par défaut dans l'utilitaire « devcpp ». Il faudra donc penser à l'activer).
Si vous avez regroupé l'ensemble des fichiers dans un seul fichier « tp3.zip », vous devez faire la remise comme suit :

1. commencez d'abord par vous connecter sur la machine « remise » comme il a été pratiqué dans la démo #01.
2. envoyez l'ensemble de vos fichiers par la procédure de remise électronique habituelle (Pour obtenir de l'aide sur cette commande, tapez dans un Xterm : man remise). Respectez les noms des fichiers.

   remise ift1166 tp3 tp3.zip

3. Vérifiez que la remise s'est effectuée correctement.

   remise –v ift1166 tp3

Barème : ce TP3 est noté sur **15 points**.

	15 points
Compilation et respect des spécifications	2
Codage, commentaires etc.	5
Rapport	4
Tests	4

En plus du précédent barème, vous risquez de perdre des points dans les cas suivants ….

- La non remise électronique (volontaire ou par erreur) est sanctionnée par la note 0.
- Les programmes ne contenant pas d'en-tête, -1 point.
- Un programme qui ne compile pas : 0.
- Un programme qui compile mais ne réalise pas les choses prévues dans la spécification : 0.
- Les avertissements (warnings) non corrigés : cela dépend de la quantité! À partir de -0.25 et plus.
- Le non respect du nom du fichier va générer une erreur de compilation donc un des points de la spécification n'a pas été respecté : 0.
- Aberration dans le codage : même si tous les chemins mènent à Rome, faites l'effort nécessaire pour éviter de prendre le plus long!

Bonus : ce bonus est noté sur **4 points**. Il vous est demandé de faire les solutionnaires de certains exercices posés dans les planches des démonstrations #10 & #11. Il ne vous est donc pas demandé de remettre un rapport, mais des programmes bien commentés (et documentés) qui compilent et s'exécutent correctement pour obtenir la note complète. Chaque exercice est noté, indépendamment, sur un point.

No de la démonstration	**No de la question**	**Barème**	**Nom du fichier à remettre**
Démonstration #10	Question #05	1 point	d10q5.cpp
Démonstration #11	Question #02 (et donc Q#01 aussi)	1 point	d11q2.cpp
Démonstration #11	Question #06	1 point	d11q6.cpp
Démonstration #11	Question #09	1 point	d11q9.cpp

Vous pouvez vous servir des solutionnaires proposés dans la démonstration #09 pour avoir une idée ce qu'il doit contenir comme informations d'un solutionnaire donné.

Ces fichiers doivent faire partie de « tp3.zip ».

Des questions à propos de ce TP?

Une seule adresse : dift1166@iro.umontreal.ca

Pour faciliter le traitement de votre requête, inclure dans le sujet de votre email, au moins la chaîne: [IFT1166] et une référence au tp03.

Mise à jour

18-06-2008 diffusion

Annexe -1-

Un exemple de séparation sur 3 fichiers

1- Rappel du processus de génération du programme exécutable
Le préprocesseur se charge d'exécuter les directives « # » introduites dans votre code. Par exemple : les macros, l'inclusion de fichiers etc.

Le Compilateur va vérifier la syntaxe et si les fonctions appelées existent effectivement. Si par exemple, vous avez appelé la méthode « sin » (pour « sinus ») et vous avez oublié d'inclure la directive « #include <cmath> » le compilateur ne saura pas à quoi va correspondre l'appel de la fonction « sin ». À ce stade, il n'a pas besoin de savoir ce que va faire la fonction « sin ». Il a juste besoin de savoir qu'elle existe donc il n'a besoin que de son prototype.

L'éditeur de liens va faire en sorte que la définition de la fonction va être chargée. On rassemble donc tous les liens (les morceaux que nous avons compilés séparément) pour arriver au programme exécutable (un bloc unique). Si cette définition a été omise, vous aurez une erreur de « linkage ».

2- Découpage
Le découpage est utile du point de vue structurel, si aussi vous voulez ne remettre que l'interface et le code objet à l'usager etc. Vous voyez mieux l'organisation de votre programme avec l'intention d'avoir une programmation orientée objet donc d'une séparation en amont.

L'interface sera donc présentée sous la forme d'un fichier « *.h » qui porte le nom de fichier d'en-tête. L'usager va prendre connaissance de l'existence des noms des classes, méthodes etc. donc les prototypes. Il n'a pas besoin de savoir comment ces classes et méthodes ont été codées.

Pour revenir à l'exemple de la fonction « sin », vous avez juste besoin de savoir qu'elle existe (son prototype) afin de pouvoir l'utiliser correctement (donc faire appel). Nous avons besoin de savoir où a été déclaré son prototype (le fichier d'en-tête qui va avec : « cmath »). La méthode de codage n'est pas votre problème. Vous allez compter sur l'éditeur de liens pour qu'il fasse son travail de recherche de la définition de cette méthode pour l'inclure dans le programme exécutable.

3- Exemple
Nous allons prendre comme exemple les trois fichiers « affiche.h », « affiche.cpp » et « main.cpp ».

- Un fichier « affiche.h » va contenir la définition de la classe (ou les classes si vous décidez de découper votre code selon plusieurs classes).

Il est nécessaire de définir la classe dans un bloc supervisé par un test via des macros pour éviter de la déclarer deux fois : d'où l'utilisation de « #ifndef », « #define », « #endif ». Comme cela déjà précisé, vous n'êtes pas limités sur le nombre de macros.

- Un fichier « affiche.cpp » va contenir la définition des méthodes des classes déclarées dans le fichier « affiche.h ».

Il faudra inclure (la directive « #include ») le fichier « affiche.h » pour que le fichier « affiche.cpp » prenne connaissance en premier lieu de l'existence de la classe « Test » ainsi que ses méthodes.

- Un fichier « main.cpp » pour tester ce découpage. Comme le fichier « main.cpp » va utiliser des méthodes de la classe « Test », il a besoin de connaître l'existence de cette classe au moment de la compilation. Pour cette raison, nous avons introduit la directive « #include "affiche.h" » au début du fichier « main.cpp ».

4- Compilation des fichiers séparés

a) « affiche.h » : il ne peut pas être compilé au sens du terme que vous avez l'habitude de faire.

b) Compilation des fichiers sources :

===> affiche.cpp:

g++ -Wall -pedantic -Os -c affiche.cpp -o affiche.o

(CTRL-F7 sous Scite)

Le compilateur va produire un fichier "objet" « affiche.o »

==> main.cpp

g++ -Wall -pedantic -Os -c main.cpp -o main.o

(CTRL-F7 sous Scite)

Le compilateur va produire un fichier "objet" « main.o »

c) Éditions de liens :

Vous devez faire cette étape manuellement (pour les pros un « makefile » peut faire l'affaire). À partir de tous les fichiers objets créés séparément nous allons tisser les liens pour générer le fichier exécutable comme suit :

g++ -o main.exe main.o affiche.o

-4- Compilation des fichiers séparés en une seule passe :

Vous pouvez effectuer les opérations de compilation et d'édition de liens en une seule opération :

g++ -Wall -pedantic -Os -o main.exe main.cpp affiche.cpp

Un inconvénient à cela, s'il y a des avertissements au moment de la compilation et des erreurs au moment du linkage, vous devez apprendre à démêler cela. Il est donc conseillé d'aller plutôt avec l'étape -3-.