Programació orientada a objectes amb C++. Herència i polimorfisme.

Pràctica 0

Estructura de Dades *– Universitat de Barcelona*

Rubén Ballester Bautista – Oriol Rabasseda Alcaide

NIUB - Nom:

16799506 – Rubén Ballester Bautista

16856733 – Oriol Rabasseda Alcaide

Grup de Pràctiques ***F*** – Parella ***1***

Professora:

Maria Salamó

Exercicis comentats

Exercici 1

En aquest cas encara no havíem vist les excepcions i per tant les tractem d'un mètode rudimentari, enviant un missatge d'error amb una estructura condicional `if/else`. En aquest exercici hem fet ús dels mètodes de control de la llibreria estàndard de retransmissió de dades per teclat *«cin»*. Concretament `cin.good()` i `cin.fail()` que retornen un booleà que permet veure si l'usuari ha introduït una dada del tipus que necessita la variable que recull la informació d'entrada.

Exercici 2

Seguint la guia proposta per l'enunciat de l'exercici, hem implementat la nostra primera classe «Triangle» a C++. A més hem fet una primera aproximació al sistema de manipulació d'errors donat per C++, llançant un missatge d'error en cas què l'usuari introdueixi una entrada no vàlida e imprimint-la al bloc *«catch»*, malgrat que les excepcions llançades encara són rudimentàries atès que no les havíem donat, com hem dit abans.

Exercici 3

En aquest cas hem ampliat l'exercici 2 per tractar diferents tipus de polígons. No te més misteri que declarar una nova classe i implementar els mètodes corresponents. Tanmateix es poden fer observacions interessants d'aquests exercicis. Per exemple, podem veure que amb poca mesura de programa, el paradigma funcional podria haver sigut més curt i clar i, a més, podem veure que l'estructura modular sense herència pot complicar les coses reduint l'eficiència degut a que s'ha d'escriure tot el codi per a les mateixes funcions sense poder declarar-hi característiques en comú, utilitzant l'herència abans citada. Mateix tractament d'errors rudimentari de l'exercici 2.

Exercici 4

El dit a l'apartat de l'exercici 3. Només cal comentar el tractament de fitxers mitjançant *stream*. Com podem veure utilitza els mateixos operadors per llegir informació que els utilitzats per *«cin» (*de fet les llibreries estàndard de *stream* de C++ en general utilitzen els operadors <<(*get*) i >> (*set*)). Per llegir dades d'un arxiu utilitzem la classe de *stream* de fitxers `<fstream>` pertanyent a les llibreries per defecte.

Exercici 5

Implementem per primera vegada excepcions com a classes. Cal comentar que la decisió presa de tractar dues excepcions en una mateixa subclasse de tipus tractament d'errors és una mala pràctica, però en aquesta ocasió ho hem fet així per experimentar amb l'herència de llibreries de sistema relacionats amb excepcions. A més per primera vegada utilitzem el concepte d'herència comentat en moltes ocasions a aquesta memòria. Com podem veure la modularitat millora considerablement utilitzant aquest concepte. Millora la qualitat del codi, en general.

Exercici 6

En aquesta ocasió hem utilitzat per primera vegada contenidors de les llibreries per defecte de C++. Concretament un vector. Com C++ no permet fer un contenidor de elements abstractes, ja que no es poden inicialitzar, ho hem fet de punters de la classe Figure, que al ser una superclasse. permet guardar objectes de subclasses de diferents tipus. Aquesta decisió ens ha conduït a implementar al destructor una seqüència que faci un cast dinàmic dels elements per identificar els tipus d'elements (ja que estan emmagatzemats com tipus *Figure*) i poder cridar als seus destructors corresponents. A més hem netejat el vector fent un clean() després de eliminar tots els objectes.

Hem necessitat utilitzar a les seqüències els *Iteradors*, que ens permeten iterar sobre un contenidor genèric sense conèixer la seva implementació interna. Hem esbrinat que un iterador és un punter que apunta a un objecte en una posició concreta i que utilitza els mateixos operadors aritmètics per l'increment o el decrement.

Qüestions:

Què ens permet fer l'herència que no podríem fer altrament?

Ens permet definir entitats amb característiques en comú i treballar amb elles. Aquest procés ens aporta diverses millores, com l'increment de la modularitat i la millora de l'encapsulació. A més a més, aquestes dues característiques essencials ens permeten un procés d'abstracció més eficient a l'hora de dissenyar un programa, necessitant un menor temps de implementació i fent una millora considerable de la reutilització de codi que no podem aprofitar en altres paradigmes de programació com la funcional o el disseny modular més «primitiu» (ja que l'herència junt amb el polimorfisme són característiques fonamentals pròpies del desenvolupament orientat a objectes).

Per exemple, a l'hora de dissenyar un software de càlcul de perímetres podem definir una classe abstracta de polígons que sigui la mare de diferents subclasses que implementin una funció virtual definida a la classe abstracta. D'aquesta manera si necessitem calcular nous tipus de polígons tan sols cal estendre la classe amb una subclasse, disminuint dràsticament la mesura del programa en relació a altres paradigmes, com hem dit abans.

Perquè els constructors i destructors els hem de cridar a les classes derivades i no a la classe base?

Segons el tipus d'objecte que sigui, si que es podria cridar al constructor de la classe base, ja que crida a la superclasse principal `Object` que s'encarrega d'eliminar el objecte en qüestió satisfactòriament. Tanmateix no és una bona pràctica ja que cada subclasse pot tindre declarats atributs que s'han de destruir individualment i cridant al mètode pare deixaríem residus a memòria, malgastant recursos.

Es podria fer que Figure implementés el mètode `getPerimeter`?

Al ser un mètode virtual d'una classe «abstracta» no és pot definir atès que mètode es defineix virtual com a mètode a implementar per les diferents subclasses. Tanmateix, C++ t'obliga a donar un retorn per defecte per si alguna subclasse no ha definit el seu propi mètode virtual heretat.

Anomena els membres de dades definits en els teus programes i digues a quina classe pertanyen. Explica les decisions de visibilitat de cadascun.

- Classe Rectangle
  - var base => double. Tipus predefinits de nombres racionals. Privat. L'usuari no necessita conèixer la implementació.
  - var alcada => double. Tipus predefinits de nombres racionals. Privat. L'usuari no necessita conèixer la implementació.
- Classe Square
  - var side => float. Tipus predefinits de nombres racionals. Privat. L'usuari no necessita conèixer la implementació.
- Classe Triangle
  - var base => double. Tipus predefinits de nombres racionals. Privat. L'usuari no necessita conèixer la implementació.
  - var alcada => double. Tipus predefinits de nombres racionals. Privat. L'usuari no necessita conèixer la implementació.
- Classe FigureContainer
  - var figureList => vector<Figure*>. Privat. L'usuari no necessita conèixer la implementació.

L'iterador que recorre les figures, quant s'incrementa cada cop? Perquè?

Suposant que ens referim a l'exercici quan tractem les figures pertanyents al vector «`FigureContainer`» podem dir que a cada passada s'incrementa una posició. Tanmateix com no coneixem la implementació del TAD contenidor vectorial (vector) no podem assegurar com fa el TAD l'increment. Açò es degut a la encapsulació de les llibreries per defecte de C++. De fet, per a tots els contenidors de la llibreria «default» de C++ la instrucció d'increment «`var++`» és funcional ja que en C++ es pot declarar que fan els operadors per a cada classe.