JP, Wtorek, 4.12.2011, 16:15-17:45

Tworzymy sklep sprzedający komputery. Przyjmujemy, że nasz sklep sprzedaje laptopy lub komputery stacjonarne (desktopy). Chcemy stworzyć aplikację, którą można rozwinąć w bazę danych posiadanego sprzętu komputerowego. Zakładamy, że laptop opisany jest przez następujące parametry: producent, model, cena, pamięć RAM, pojemność HDD, rozdzielczość matrycy (liczba linii pikseli w pionie, np. „1080", „720"), rozmiar matrycy w calach, zaś desktop jest opisany przez: producent, model, cena, pamięć RAM, pojemność HDD, czy posiada monitor, czy posiada mysz i klawiaturę. Zauważmy, że laptop i desktop mają 5 wspólnych parametrów: producent, model, cena, pamięć RAM, pojemność HDD. By zatem zaoszczędzić sobie pracy przy pisaniu możemy wykorzystać najważniejszą bodajże własność programowania obiektowego – **dziedziczenie**, by stworzyć klasę nadrzędną, np. `Komputer`, która by zawierała pola i metody, które byłyby wspólne dla klas pochodnych: `Laptop` oraz `Desktop`. W dodatku, gdyby nasz sklep chciał poszerzyć asortyment i wprowadzić sprzedaż np. tabletów, tudzież innych komputerów, nie stanowiłoby to problemu – należałoby napisać kolejną klasę dziedziczącą z klasy `Komputer`.

Do wykonania:

1. Napisać klasę `Komputer` o następujących polach składowych:
```
std::string fProducent;
std::string fModel;
double fCena;
int fRAM; //w GB
int fHDD; //w GB
```
zawierającą również następujące metody:
```
void SetProducent(std::string producent);
void SetModel(std::string model);
void SetCena(double cena);
void SetRAM(int ram);
void SetHDD(int hdd);
std::string GetProducent();
std::string GetModel();
double GetCena();
int GetRAM();
int GetHDD();
```
2. Oraz klasy `Laptop` i `Desktop`, dziedziczące z klasy `Komputer`:
   - Klasa `Laptop` powinna posiadać dodatkowe pola:
```
double fRozmiarMatrycy; //w calach
int fRozdzielczoscMatrycy; //liczba linii pikseli
```
dwa konstruktory:
```
Laptop();
Laptop(std::string producent, std::string model, double cena, int ram, int hdd, double rozmiar, int rozdzielczosc);
```
oraz metody:
```
void SetRozmiarMatrycy(double rozmiar);
void SetRozdzielczoscMatrycy(int rozdzielczosc);
double GetRozmiarMatrycy();
int GetRozdzielczoscMatrycy();
```
   - Klasa `Desktop` powinna posiadać dodatkowe pola:
```
bool fMonitor;
bool fMyszKlawiatura;
```
dwa konstruktory:
```
Desktop();
Desktop(std::string producent, std::string model, double cena, int ram, int hdd, bool monitor, bool
```

```
myszklawiatura);
```
oraz metody:
```
void SetMonitor(bool monitor);
void SetMyszKlawiatura(bool myszklawiatura);
bool CzyMonitor();
bool CzyMyszKlawiatura();
```
Trzeba zauważyć, że wszystkie klasy i metody są bardzo podobne – posiadają jedynie funkcje typu „get" i „set" oraz, w przypadku klas `Laptop` i `Desktop`, dwa konstruktory (domyślny i z parametrami). Zadaniem funkcji typu „set" jest ustawienie odpowiednich pól składowych, tzn.:
załóżmy, że mamy w naszej klasie pole double `fCena`. Wtedy metody „set" i „get" dla takiego pola:
```
void SetCena(double cena) //metoda typu "set"
{
  fCena = cena;
}
int GetCena() //metoda typu "get"
{
  return fCena;
}
```
Aby wykonać zadanie:

1. Tworzymy osobny katalog, w którym tworzymy 7 pustych (na początku) plików: **Komputer.h**, **Komputer.cpp**, **Laptop.h**, **Laptop.cpp**, **Desktop.h**, **Desktop.cpp** oraz **program.cpp**
   (w pliku **program.cpp** tworzymy fod razu unkcję `int main()` zwracającą 0).
2. Tworzymy ósmy plik, prosty plik tekstowy o nazwie **Makefile**, do którego należy wpisać:
   ```
   all:
      g++ -Wall Komputer.cpp Laptop.cpp Desktop.cpp program.cpp
      -o program
   ```
   To pozwoli nam korzystać z polecenia `make` w linii poleceń i kompilować program wpisując w linii poleceń jedynie `make`. Próbujemy skompilować program.
3. Tworzymy klasę `Komputer`: **1 p.**
   - zaczynamy od pliku nagłówkowego **Komputer.h** tworząc definicję klasy i deklarując wszystkie pola i metody (pamiętamy o średniku na końcu definicji klasy!).
   - Do pliku **Komputer.cpp** wpisujemy definicje wszystkich metod klasy `Komputer`.
   - W pliku **program.cpp** w funkcji `main()` towrzymy obiekt typu `Komputer` i testujemy, czy wszystko działa dobrze używając metod klasy `Komputer`, np.:
     ```
     Komputer p;
     p.SetProducent("ASUS");
     cout<<"Producent komputera p: "<<p.GetProducent()<<endl;
     ```
   - Kompilujemy program (`make`) i uruchamiamy – sprawdzamy, czy działa.
4. Tworzymy klasę `Laptop`: **3 p.**
   - Klasa ta jest pochodną klasy `Komputer`. Klasy pochodne definiuje się w następujący sposób (dla przykładu klasy podrzędnej `A` dziedziczącej z klasy `B` nadrzędnej):
     ```
     class A : public B {...pola, konstruktory, metody...};
     ```
   - Deklarujemy wszystkie pola, konstruktory i metody klasy w pliku **Laptop.h**.
   - W pliku **Laptop.cpp** wpisujemy wszystkie definicje konstruktorów oraz pól (dla testu, w pierwszej fazie, można konstruktorów nie wpisywać).
   - W pliku program.cpp tworzymy obiekt typu `Laptop` i sprawdzamy, używając konstruktorów i metod, czy wszystko działa.
5. Klasę `Desktop` tworzymy analogicznie jak klasę `Laptop`, sprawdzamy czy wszystko działa. **1 p.**

Kilka ważnych uwag!

1. Czytamy DOKŁADNIE błędy kompilatora i zawsze zaczynamy je poprawiać od pierwszego (czyli od góry).
2. Błędy typu:
   `error: redefinition of ‘class Komputer’`
   `error: previous definition of ‘class Komputer’`
   oznaczają, że wiele razy wstawiliśmy include naszej klasy i kompilator widzi wiele razy jej definicję. Od tych zajęć ZAWSZE pliki nagłowkowe (**.h**) będziemy „otaczać” następującymi dyrektywami `define` (takie operacje nazywamy tokenem) np.:

```
#ifndef KlASA_H
#define KLASA_H

class Klasa {...pola, konstruktory, metody...};

#endif //KLASA_H
```

   Działa to w ten sposób, że jeśli token `KLASA_H` nie był jeszcze zdefiniowany, to kompilator wejdzie do środka i zrobi wszystko między `ifndef` a `endif`. Jeśli był zdefiniowany, to od razu przejdzie do `endif`. W ten sposób nie musimy się już przejmować kolejnością ładowania plikół nagłówkowych – jeśli były gdzieś załadowane wcześniej, to kompilator je pominie.
3. Błędy typu:
   `'int Pojazd::fModel' is private`
   oznaczają, że wszystkie pola typu private są niedostępne poza obszarem danej klasy, tym RÓWNIEŻ dla klas pochodnych. Często jednak chcemy (a na pewno chcemy w naszym przypadku), by pola klasy nadrzędnej były dostępne w klasie pochodnej ale nie były dostępne poza klasami. W tym celu należy użyć kwalifikatora dostępu typu `protected`. Pola, które w klasie nadrzędnej opatrzone są takim kwalifikatorem, są dostępne w klasach pochodnych ale nie są dostępne poza klasami.