Języki Programowania, Zadanie 7 – realizowane w parach

Dziedziczenie

Dziedziczenie jest to technika pozwalająca na definiowanie nowej klasy, przy wykorzystaniu klasy już wcześniej istniejącej. Często się zdarza, że mamy kilka klas podobnych do siebie. Możemy wtedy stworzyć klasę podstawową, zawierającą część wspólną wszystkich klas, a następnie tworzyć klasy pochodne, zgodnie ze zdaniem „Chcę mieć taką samą klasę jak moja klasa podstawowa, z małymi różnicami (zwykle: dodatkami). Różnice te podaję poniżej...".

Dziedziczenie = Tworzenie klas pochodnych

Treść zadania

Tworzymy sklep sprzedający używany sprzęt elektroniczny. Na początek przyjmijmy, że sprzedajemy tylko dwa typy sprzętu: telewizory (o składnikach nazwa, wiek, cena i przekątna) oraz mikrofalówki (o składnikach nazwa, wiek, cena, pojemność oraz moc). Zamiast tworzyć dwóch całkowicie oddzielnych klas zauważamy, że dla obu sprzedawanych przez nas rodzajów sprzętu część składników się pokrywa (mianowicie: nazwa, wiek oraz cena). W dodatku nasza firma zastanawia się nad rozpoczęciem sprzedaży odtwarzaczy mp3, dla których owe trzy składniki również by się pokrywały. By zaoszczędzić na pisaniu tego samego dwa razy (oraz oszczędzić pracy w przyszłości) postanawiamy napisać klasę nadrzędną Towar z której klasy Telewizor oraz Mikrofalówka będą dziedziczyć.

Należy napisać klasę **Towar** o następujących polach składowych

- `std::string nazwa;`
- `int wiek;`
- `double cena;`

konstruktorem bezparametrowym, konstruktorem z 3 parametrami, oraz metodach:

- `SetNazwa(std::string n)`
- `SetCena(double c)`
- `SetWiek(int w)`
- `std::string GetNazwa()`
- `double GetCena()`
- `int GetWiek()`
- `void Wypisz()`

Oraz klasy:

- **Telewizor** dziedziczącą z Towaru, z dodatkowym polem `double przekatna`. Z konstruktorami: domyślnym (zerującym wszystkie pola) oraz konstruktorem z czterema parametrami. Ponadto klasa telewizor powinna mieć metody: `„Wypisz”` (bezargumentową, wypisującą dane telewizora na ekran) oraz `double GetPrzekatna()` i `void SetPrzekatna(double p)`. **Nie używamy listy inicjalizacyjnej konstruktora.**

- **Mikrofalówka** dziedziczącą z Towaru, z dodatkowymi polami „int pojemność” (w litrach), oraz „int moc” (w kW). Z konstruktorami: domyślnym (zerującym wszystkie pola), z trzema parametrami (podstawowymi dla klasy Towar, czyli nazwą, wiekiem i ceną, reszta jest zerowana) oraz z pięcioma parametrami. **Należy użyć listy inicjalizacyjnej konstruktora do ustawienia pól składowych w konstruktorach.**

Trzeba zauważyć, że wszystkie klasy i metody są bardzo podobne – posiadają jedynie funkcje typu „get” i „set” oraz, konstruktory (domyślny i z parametrami).

W głównej funkcji programu należy stworzyć

- obiekt klasy `Towar` przy użyciu konstruktora z 3 parametrami.
- po trzy obiekty klasy `Telewizor`, przy użyciu różnych konstruktorów,
- trzy obiekty klasy `Mikrofalowka`, przy użyciu różnych konstruktorów.
- Następnie należy stworzyć po trzy wskaźniki na takie obiekty, również przy użyciu różnych konstruktorów.

Należy postępować według punktów:

1. Należy stworzyć osobny katalog, a w nim 5 pustych plików tekstowych: **program.cpp**, **towar.cpp**, **towar.h**, **telewizor.cpp**, **telewizor.h**. Następnie należy stworzyć **Makefile**. W pliku program.cpp należy dodać funkcję `int main()` zwracającą 0.
   Należy skompilować tak przygotowany program używając w linii poleceń komendy: `make.`
2. Następnie tworzymy klasę `Towar`.
3. Tworzymy klasę `Telewizor`.
   - Tworzymy klasę Telewizor, będącą klasą pochodną klasy Towar
     ```
     class nazwa_klasy_pochodnej : public nazwa_klasy_podstawowej {};
     ```
   - Zapisujemy, kompilujemy. Poprawiamy błędy. Wskazówki:
     a) ZAWSZE poprawiamy najpierw pierwszy błąd na liście.
     b) W przypadku błędu typu:
     ```
     telewizor.h:5:7: error: redefinition of ‘class telewizor’
     telewizor.h:5:12: error: previous definition of ‘class telewizor’
     ```
     ten błąd występuje, wtedy gdy dwa razy próbujemy dodać tę samą klasę. Kompilator się skarży, że próbujemy drugi raz zdefiniować to samo. Prawidłową metodą radzenia sobie z tym jest owijanie definicji klas w plikach nagłówkowych w strukturę „ifndef”:
     ```
     #ifndef _NAZWA_TOKENU
     #define _NAZWA_ TOKENU
        class klasa{...}; - definicja naszej klasy
     #endif
     ```
     c) W przypadku błędu typu:
     ```
     ‘int komputer::wiek’ is private
     ```
     Domyślnie wszystkie zmienne **`private`** są niedostępne poza daną klasą, czyli RÓWNIEŻ dla klas pochodnych. W naszym wypadku chcielibyśmy, by te zmienne były dla nas dostępne, ale z drugiej strony - dostępne *tylko* w naszych klasach pochodnych – nie publicznie dostępne na zewnątrz. Takie zmienne powinny zostać zadeklarowane jako **`protected`** w klasie pojazd. Tak więc tutaj wyjaśnia się, do czego służy kwalifikator dostępu `protected` - pola, które w klasie nadrzędnej opatrzone są takim kwalifikatorem, są dostępne w klasach pochodnych ale nie są dostępne poza klasami.
4. Tworzymy klasę `Mikrofalowka`. W dwóch kolejnych plikach.

5. Należy ponadto zapoznać się z treścią pliku: http://www.if.pw.edu.pl/~majanik/data/JP/2012/makefile.pdf.
   Napisany **Makefile**, powinien umożliwiać kompilację napisanego programu, definiować zmienną `CXX` określającą kompilator (g++) oraz `CXXFLAGS` definiującą flagę (-Wall), a tworzone klasy powinny być wstępnie kompilowane do plików *.o.
   Należy tak skonfigurować Geany, by móc kompilować przez cegiełkę przy użyciu stworzonego makefile’a (wystarczy raz wybrać z rowzijanej listy przy cegiełce „Make all”, by ta opcja wskoczyła jako domyślna).

6. Zamienić klasę Towar w klasę abstrakcyjną (ATD, abstrakcyjny typ danych). Metodą czysto wirtualną powinna zostać metoda Wypisz:
   - **Wypisz() = 0**

Następnie poprawiamy program tak, by się skompilował – czego nie możemy wtedy w programie zrobić?