

Laboratorium 8, Języki Programowania, 2.12.201

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.

INSTRUKCJA POSTĘPOWANIA ZNAJDUJE SIĘ NA NASTĘPNEJ STRONIE, poniżej jedynie podsumowanie:

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)`.
- **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.

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 typu **towar** i użyć na nim wszystkich napisanych metod (testujemy).
- Trzy obiekty klasy **Telewizor**, przy użyciu trzech różnych konstruktorów. Używamy na jednym z nich metody „GetPrzekatna”, wszystkie wypisujemy na ekran.
- Trzy wskaźniki na obiekty **Telewizor** (wraz z obiektami), również przy użyciu różnych konstruktorów, też wypisujemy je na ekran.
- Tworzymy obiekty klasy **Mikrofalówka** przy użyciu napisanych konstruktorów, wypisujemy je na ekran.

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. W funkcji głównej:
 - [main] Tworzymy obiekt klasy towar w głównej funkcji programu.
 - [main] Używamy na nim napisanych przez nas metod.
3. Tworzymy klasę telewizor. Po kolei:
 - Tworzymy klasę telewizor, będącą klasą pochodną klasy towar

```
class nazwa_klasy_pochodnej : public nazwa_klasy_podstawowej {};
```
 - Deklarujemy dodatkowe składniki, konstruktory oraz metody.
 - [main] W głównym programie tworzymy obiekt klasy telewizor, następnie wypisujemy go na ekran, używając metody `Wypisz()`.
 - Zapisujemy, kompilujemy. Poprawiamy błędy. Wskazówki:
 - a) W przypadku błędu typu:

```
towar.h:5:7: error: redefinition of 'class towar'
towar.h:5:12: error: previous definition of 'class towar'
```

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
```
 - b) W przypadku błędu typu:

```
'int towar::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 towar. 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. Zamieniamy konstruktory dla klasy telewizor w ten sposób, by użyta została **lista inicjalizacyjna konstruktora**.
5. Tworzymy klasę mikrofalówka. Postępujemy dokładnie tak samo jak z klasą telewizor. Dla mikrofalówki **nie używamy listy inicjalizacyjnej konstruktora**.
6. Następnie **zamieniamy metodę wirtualną z klasy podstawowej na czysto wirtualną**

```
virtual void Wypisz() = 0;
```

i poprawiamy program tak, by się skompilował – czego nie możemy wtedy w programie zrobić?
Wskazówka: wszystkie metody czysto wirtualne muszą istnieć w klasach pochodnych.
Klasa, która ma co najmniej jedną funkcję czysto wirtualną nazywamy klasą **abstrakcyjną**.

Po co tworzyć klasy abstrakcyjne?

Czasem mamy jakiś obiekt, który łączy cechy kilku innych (jak np. nasza klasa Towar) ale sama nie przedstawia swojej istotnej żadnego konkretnego obiektu. Klasa abstrakcyjna jest klasą jakby nieskończoną. Jej dokończenie realizowane jest przez klasy pochodne.

Należy zwrócić jeszcze uwagę na pewną zasadę, którą należy stosować:

Jeśli klasa deklaruje jedną ze swoich funkcji jako virtual, wówczas jej destruktor deklarujemy także jako virtual.
Skoro w klasie deklarujemy jakąś funkcję wirtualną, to znaczy, że na obiekty klas pochodnych zamierzamy czasem mówić jak na obiekty klasy podstawowej, co przy późniejszym niszczeniu obiektów mogłoby być problemem – nie zwalniałybyśmy pamięci dla niektórych składników klas pochodnych.

7. Należy napisać odpowiedni Makefile dla tego programu. Jeśli ktoś wcześniej nie miał do czynienia z tym wyjaśnieniem, to należy ponadto zapoznać się z treścią pliku:
<http://www.if.pw.edu.pl/~majanik/data/JP/2012/makefile.pdf>.