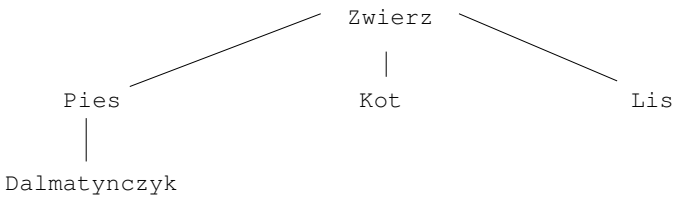
Języki Programowania, 14.12.2021
Zadanie 8
Dziedziczenie
Dziedziczenie jest to technika pozwalająca na definiowanie nowej klasy, przy wykorzystaniu klasy już wcześniej istniejącej. Często się zdarza, że mamy kilka klas podobnych do siebie. Możemy wtedy stworzyć klasę podstawową, zawierającą część wspólną 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

Wirtualność
Wirtualność oraz polimorfizm (tu rozumiany jako możliwość wyboru postaci funkcji w trakcie działania programu) stanowią zupełnie inne podejście do programowania niż programowanie proceduralne, które do tej pory wykorzystywaliśmy. Są najważniejszą cechą programowania orientowanego obiektowo (OOP – Object Oriented Programming, inaczej mówiąc: **orientującego się według typu obiektu**) takiego jak C++ i mają ogromne możliwości, które objawiają się w pełni przy dużych projektach.

Dzisiejszy program będzie demonstrować ideę i zasadę działania wirtualności. W tym celu stworzymy pewną hierarchię dziedziczenia klas. Ilustruje to schemat:



Najbardziej podstawową klasą w naszym układzie jest klasa `Zwierz`. Oznacza to, że wszystkie elementy klasy `Zwierz` będą wspólne dla pozostałych, bardziej wyspecjalizowanych klas pochodnych. Załóżmy dalej, że wspólnym parametrem opisującym wszystkie zwierzęta jest masa zwierzęcia, zatem możemy w klasie `Zwierz` wprowadzić pole `double fMasa` – domyślnie = 0 (oraz metody set i get ustawiające i pobierające to pole).
Cały program należy napisać w trzech plikach (klasa.cpp, klasa.h dla wszystkich klas łącznie oraz main.cpp).

Oprócz tego chcemy wprowadzić dwie proste metody, w tym pierwsza z nich powinna być metodą wirtualną:
```
virtual void Zwierz::PrzedstawSie()
{      cout<<"Zwierz: ..."<<endl;
}
string Zwierz::CzymJestem()
{      return "jestem zwierzeciem";
}
```

W zasadzie w klasie tej nie ma nic szczególnego, poza słowem kluczowym "**`virtual`**" przed funkcjami składowymi w klasie `Zwierz`.

Stwórzmy jeszcze dwie krótkie klasy, będące klasami pochodnymi klasy Zwierz: klasy `Pies` oraz `Kot`. Domyślnie pies waży 10 kg, a kot 3 kg (odpowiednie konstruktory!). Klasy te powinny posiadać tylko po jednej funkcji składowej: `PrzedstawSie()` (już nie wirtualnej), np.:
```
void PrzedstawSie()
{ cout<<"Kot: Miau Miau!"<<endl;
}
```
W funkcji `main` stwórzmy teraz nasz zwierzyniec: po jednym obiekcie klasy `Zwierz`, `Pies` oraz `Kot` (np. `zwierzak`, `azor`, `mruczek`), następnie na każdym z tych obiektów należy wywołać metodę `PrzedstawSie()`.
Należy program odpalić i skompilować. Zachowanie użytych funkcji nie powinno być zaskakujące.

Następnie należy stworzyć pojedynczy wskaźnik na klasę `Zwierz`.
```
Zwierz* wskzwierz;
```
i przypisać do niego po kolei wszystkie nasze zwierzątka oraz wywołać na takim wskaźniku metodę `PrzedstawSie`, np.:
```
wskzwierz = &zwierz1;
wskzwierz->PrzedstawSie();
```

Program należy ponownie skompilować i odpalić. Co się dzieje, jeśli usuniemy słowo `virtual` z klasy `Zwierz` (należy przetestować i opowiedzieć prowadzącemu)? **1.5 p**

Teraz stwórzmy metodę globalną : **1 p.**
`void DajGlos(Zwierz& zwierzak)`, której jedynym zadaniem będzie wywołanie metody `PrzedstawSie` na referencji `zwierzak`. Przypomnijmy tutaj, że referencja oznacza „przezwisko" danego obiektu. W jaki sposób teraz zareaguje kompilator?

Jakie daje nam to możliwości?
- Możemy tworzyć metody przyjmujące referencje, których zachowanie będzie zależne od typu klasy pochodnej.

- Jeśli pojawi się kiedykolwiek konieczność rozszerzenia programu o nowe obiekty, to dodanie nowej klasy będącej pochodną od już istniejącej nie wymaga tysiąca zmian w różnych miejscach programu.
Należy dodać klasę `Lis`, zachowującą się podobnie do poprzednich dwóch, wywołać na niej gotową już metodę `DajGlos` . Prawda, że proste? (jakby ktoś się zastanawiał, jakie dźwięki wydaje lis :) *„What does the fox say?”* https://www.youtube.com/watch?v=jofNR_WkoCE :)

Podsumujmy:
Jeśli kompilator natrafi na **obiekt** danej klasy, np. `Zwierz`, uruchamia dla niego funkcję składową z właściwej mu klasy.
Jeśli kompilator natrafi na **wskaźnik lub referencję** klasy `Zwierz`, ma dwie możliwości wywołania funkcji składowej:
1) Skoro jest to wskaźnik do klasy `Zwierz`, wywołuje metodę składową klasy `Zwierz`. Jest to domyślne zachowanie kompilatora.
2) Kompilator widzi, że jest to wskaźnik do klasy `Zwierz`, ale zamiast na ślepo sięgać do klasy `Zwierz` *używa swojej inteligencji*: nie daje się zwieźć typem i sprawdza, na co faktycznie wskaźnik wskazuje. Wtedy może się zorientować, że wskaźnik tak naprawdę wskazuje na obiekt klasy pochodnej `Kot`, zatem **orientując się według typu obiektu** uruchamia funkcję składową kotka. Takie zachowanie wymusza słowo kluczowe `virtual` przed nazwą funkcji składowej.

Na koniec zamieniamy metody wirtualne z klasy podstawowej na czysto wirtualne i poprawiamy program tak, by się skompilował – czego nie możemy wtedy w programie zrobić? `virtual void PrzedstawSie() = 0;`
Wskazówka: wszystkie metody czysto wirtualne muszą istnieć w klasach pochodnych.
Klasa, która ma co najmniej jedną funkcję czysto wirtualną nazywamy klasą **abstrakcyjną (abstrakcyjnym typem danych - ATD)**.

Po co tworzyć klasy abstrakcyjne?
Czasem mamy jakiś obiekt, który łączy cechy kilku innych (jak np. nasza klasa `Zwierz`) ale sam nie przedstawia swoją istotną żadnego *konkretnego* obiektu. Mamy psy, koty, krowy i inne – jak by mógł zareagować malarz, gdybyśmy kazali namalować mu zwierzę? Jakie odgłosy zwierzę takie powinno wydawać? Klasa abstrakcyjna jest klasą jakby „niedokończoną”. 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 zwalnialibyśmy pamięci dla niektórych składników klas pochodnych.

To na koniec jeszcze tylko małe powtórzenie z dziedziczenia: drugi poziom dziedziczenia.

Stwórzmy klasę `Dalmatynczyk` dziedziczącą z `Psa`. **1 p.**
Niech zawiera tylko jedną metodę: `string CzymJestem(),` zwracającą odpowiedni tekst. Następnie w funkcji `main()` należy wywołać dwie funkcje: `PrzedstawSie` oraz `CzymJestem` (oczywiście z odpowiednim przekierowaniem na standardowe wyjście, jeśli potrzeba).

Należy dopisać do klasy `Zwierz` metodę wirtualną `Zapisz`: **1.5 p.**

- Metoda `Zapisz(char*)` powinna zapisać do pliku, którego nazwa podana jest jako jej parametr odgłos jaki wydaje dany zwierzak.

Zapisywanie do pliku (przykład):

```
#include <fstream>

ofstream ofile;
ofile.open("file.txt");
ofile<<”Jestem zwierzeciem”<<endl;
ofile.close();
```

W funkcji `main` należy „zapisać” ostatnio „stworzonego psa” do pliku, którego nazwa jest podana jako **pierwszy parametr wywołania programu**. Należy mu również ustawić masę zwerzęcia podaną jako **drugi parametr** wywołania programu (uwaga, program powinien wypisywać odpowiednie ostrzeżenie, jeśli liczba podanych parametrów jest nieprawidłowa!)

Parametry wywołania programu (zmiany w deklaracji funkcji main):

```
int main(int argc, char **argv){ }
```

Gdzie:
`argc` - liczba parametrów, z którymi został wywołany program
`**argv` - tablica parametrów (każdy jako `char*`):
`argv[0]` - nazwa programu
`argv[1], argv[2]`... - kolejne parametry wywołania programu
Np. wywołanie programu program mogło by mieć formę:
`./program Ala 2 3`
Wtedy:

```
argc == 4
argv[0] == "program"
argv[1] == "Ala"
argv[2] == "2"
argv[3] == "3"
```
|| zmiana tekstu ”3” na liczbę 3 – funkcja atoi (lub atof). np int a = atoi(”3”); z biblioteki <stdlib.h>

Przyporządkowanie tych parametrów wykonuje się samoistnie w momencie wywołania programu, jedynym wkładem programisty jest zadeklarowanie funkcji `main` w formie `int main(int argc, char **argv)`

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.

Wskazówki:

a) ZAWSZE poprawiamy najpierw pierwszy błąd na liście.

b) Tworzenie klasy pochodnej będącą klasą pochodną klasy Komputer

```
class nazwa_klasy_pochodnej : public nazwa_klasy_podstawowej {};
```

c) W przypadku błędu typu:

```
komputer.h:5:7: error: redefinition of ‘class komputer’
komputer.h:5:12: error: previous definition of ‘class komputer’
```

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
```

d) 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 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.