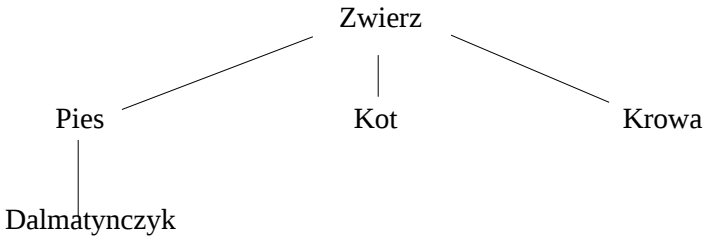
Języki Programowania, Laboratorium z dnia 11.12.2012

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) w języku C++ (inaczej mówiąc: **orientującego się według typu obiektu**) i mają ogromne możliwości, które objawiają się w pełni przy dużych projektach. To z tego powodu język ten jest wykorzystywany w pisaniu wielkich aplikacji, od systemów operacyjnych po gry komputerowe.

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



Dla przejrzystości dydaktycznej i z uwagi na fakt, że klasy będą bardzo proste i krótkie, napiszemy wszystko niestandardowo w jednym pliku program.cpp. 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 waga, zatem możemy w klasie `Zwierz` wprowadzić pole `double *fWaga` – domyślnie = 0 (oraz metody „`void set(double a)`” i „`double get()`” ustawiające i pobierające to pole). UWAGA: dla przećwiczenia wskaźników pole fWaga ma być zadeklarowane jako wskaźnik na `double`! (należy pamiętać o alokowaniu i zwalnianiu pamięci dla tego pola).

Oprócz tego chcemy wprowadzić dwie proste metody, w tym pierwsza z nich powinna być metodą wirtualną:

```
virtual void PrzedstawSie()
{     cout<<”Zwierz: ...”<<endl;
}
string 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. `zwierz1`, `pies1`, `kot1`), 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)?

Teraz stwórzmy metodę globalną :
`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 Krowa, zachowującą się podobnie do poprzednich dwóch, wywołać na niej gotową już metodę `DajGlos` . Prawda, że proste?

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 kluczwe 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 wirutalne 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 Zwierz) ale sama 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 nieskończoną. Jej dokończenie realizowane jest przez klasy pochodne.

Należy zwrócić jeszcz 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 niekórych składników klas pochodnych.

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

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

Należy ponadto zapoznać się z treścią pliku: http://www.if.pw.edu.pl/~majanik/data/JP/2012/makefile.pdf oraz zgodnie z zawartymi instrukcjami przygotować plik **Makefile** dla programu z poprzednich zajęć.