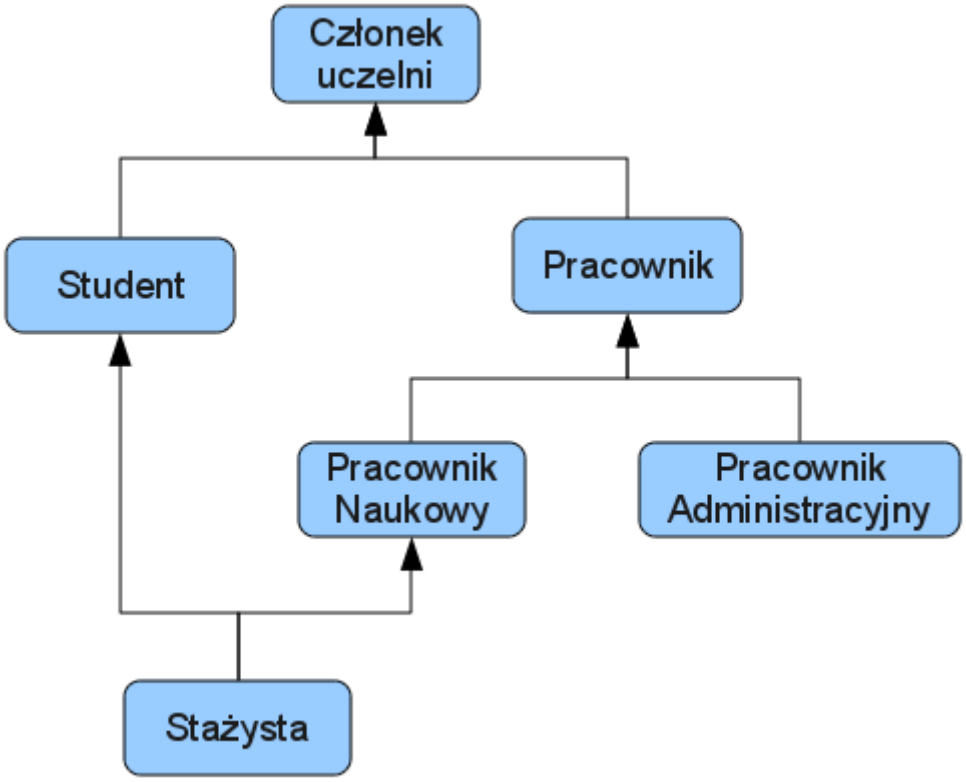
Języki i metody programowania 2

Lab 7

Dziedziczenie

Dziedziczenie to jeden z najważniejszych mechanizmów programowania obiektowego. Polega on na ponownym wykorzystaniu kodu w taki sposób, że nowe klasy tworzone są na podstawie już istniejących dziedzicząc jej metody i pola i jednocześnie dodając nowe metody i nowe pola.



Aby zapisać relację dziedziczenia w C++ stosuje się następujący schemat:

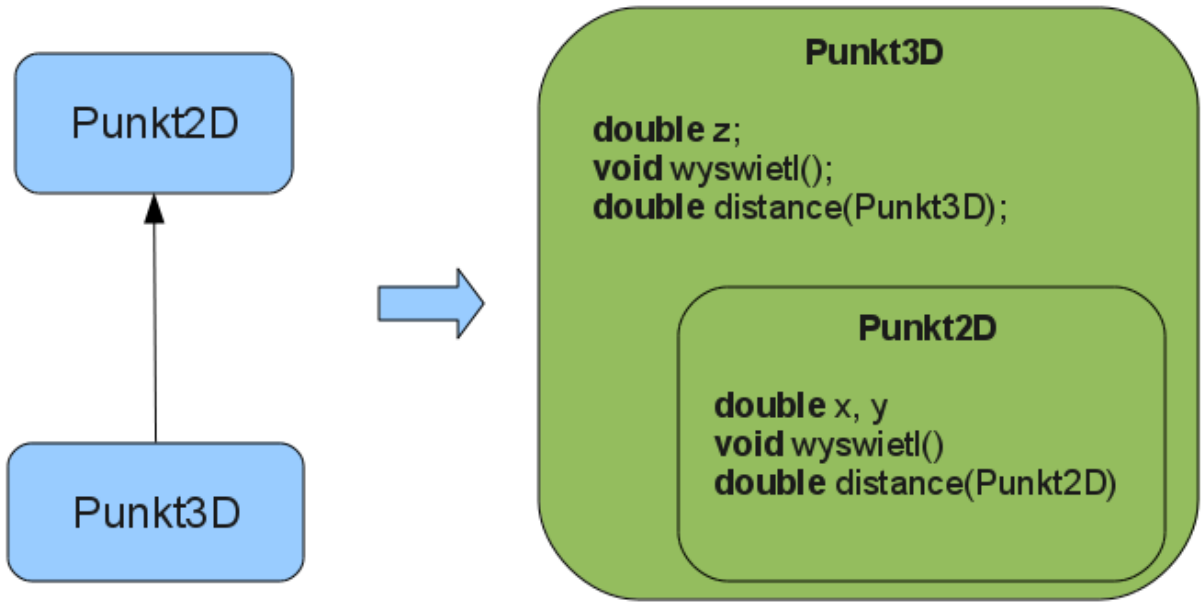
class *NazwaKlasyPochodnej* **:** *Tryb dziedziczenia NazwaKlasyBazowej* { *definicja klasy pochodnej* }

Chcąc zapisać W C++ relację z rysunku, moglibyśmy to zrobić w następujący sposób:

```
class CzlonekUczelni{...};
class Student: public CzlonekUczelni{...};
class Pracownik: public CzlonekUczelni{...};
class PracownikNaukowy : public Pracownik{...}
class PradownikAdministracyjny: public Pracownik{...};
//Dziedziczenie wielokrotne
class Stażysta : public Student, PracownikNaukowy {...};
```

Konstruktory i destruktory

Klasa pochodna tak naprawdę jest jednocześnie klasą bazową, dlatego nie można utworzyć jej obiektu bez wcześniejszego utworzenia obiektu klasy bazowej. Każdy konstruktor klasy pochodnej w pierwszej kolejności będzie wywoływał konstruktor klasy bazowej.



Konstruktora klasy bazowej nie można wywołać w ciele konstruktora klasy pochodnej, ponieważ klasy pochodne dziedziczą pola klasy podstawowej, a wszystkie pola klasy muszą zostać utworzone **przed** utworzeniem danej klasy.
Chcąc wywołać konstruktor klasy Punkt2D przez konstruktor klasy pochodnej Punkt3D stosujemy następujący zapis:

```
#include "Punkt2D.h"
#include "Punkt3D.h"

Punkt3D::Punkt3D(double x, double y, double _z) : Punkt2D(x,y) {
  z = _z;
}
```

Rzutowanie

Rzutowanie to inaczej konwersja pomiędzy typami. Wyróżnia się dwa rodzaje rzutowania:

- rzutowanie w górę (z klasy pochodnej na klasę bazową)
- rzutowanie w dół (z klasy bazowej na klasę pochodną)

Rzutowanie w górę

Rzutowanie w górę można wykonywać niejawnie, ponieważ każdy obiekt klasy pochodnej *jest* jednocześnie obiektem klasy bazowej:

```
void foo(Punkt2D point){
  cout << "Punkt ";
  point.doSomething();
}

int main(){
  Punkt3D point(12,4,5);

  foo(point);  //nastąpi rzutowanie w górę

}
```

Rzutowanie w dół

Rzutowanie w dół oznacza rzutowanie z klasy bazowej na pochodną. Jeśli nie zostanie przeciążony odpowiedni operator rzutowania, operację taką należy wykonywać przy użyciu jednego z poniższych operatorów:

Operator	Opis
static_cast<nowy_typ>(wyrazenie)	Sprawdzanie poprawności typów podczas rzutowania, wykonywane jest podczas kompilacji.
const_cast<nowy_typ>(wyrazenie)	Może być stosowany tylko ze wskaźnikami lub referencjami. Pozwala na anulowanie stałości wskaźnika obiektu (*const*).
dynamic_cast<nowy_typ>(wyrazenie)	Może być stosowany tylko ze wskaźnikami lub referencjami. Pozwala na rzutowanie w górę (dla wszystkich klas) i na rzutowanie w dół (tylko dla klas polimorficznych), sprawdzając czy obiekt który rzutujemy jest faktycznie kompletnym obiektem klasy na którą chcemy go rzutować. Operator przeprowadza ten test podczas działania programu. Jeśli konwersja nie może się odbyć, zwracany jest wskaźnik null (w przypadku referencji rzucany jest wyjątek *bad_cast*)
reinterpretc_cast<nowy_typ>(wyrazenie)	Stosowanie tego operatora jest potencjalnie bardzo niebezpieczne. Może on rzutować dowolny typ na dowolny inny typ, nawet jeśli oba typy nie mają ze sobą nic wspólnego.

Poniżej przedstawiono przykłady użycia operatorów i różnice w ich stosowaniu:

```
class A {
  public:
    void foo() = 0;
};

class B: public A{
  public:
    void foo(){}
};

class C {
  public:
    void bar(){};
};

class D: public C{
  public:
    void otherbar(){}
};

int main(){
  // static_cast - zwykłe rzutowanie w górę
  D* ptrD = new D();
  C* ptrC = static_cast<C*>(ptrD);

  // const_cast - pozbywamy się modyfikatora const
  const C* constPtrC = new C();
  C* ptrCC = const_cast<C*>(constPtrC);

  // dynamic_cast - rzutowanie w dół
  A* ptrA = new B();
  B* ptrB = dynamic_cast<B*>(ptrA);
```

```
  // reinterpret_cast - rzutowanie pomiedzy kompletnie niepowiazanymi klasami
  B* ptrBB = new B();
  D* ptrDD = reinterpret_cast<D*>(ptrBB);
}
```

Zadania

1. Napisz dwie klasy: Circle (Koło) i Sphere (Kula). Klasa Circle powinna być klasą bazową dla klasy Sphere.
    a. W klasie Circle powinny znaleźć się następujące pola i metody:
        i. **double** x, y, r - określające odpowiednio współrzędne środka koła i jego promień.
        ii. **double** pole() - obliczająca pole koła
    b. W klasie Sphere powinny znaleźć się następujące pola i metody:
        i. **double** z - określająca współrzędną przestrzenną środka kuli
        ii. **double** pole() - obliczająca pole powierzchni kuli
2. Zdefiniuj klasę **Serializable** zawierającą pojedynczą metodę **Serialize (Serializer *)**. Każda klasa implementująca go zapisuje swój stan do obiektu **Serializer** pole po polu. Zdefiniuj również dwa różne formaty serializacji. Klasy **XmlSerializer** i **JsonSerializer** umożliwiające zapisywanie obiektów w formacie odpowiednio xml lub json.