Programowanie obiektowe w języku C++

Stanisław Gepner

sgepner@meil.pw.edu.pl

Dziedziczenie

Wstęp

Zacznijmy od przykładu, w którym dziedziczenie by się przydało, ale go jeszcze nie użyjemy.

- Przykład rolniczy
- Każde zwierzątko wydaje dźwięk: *metoda dźwięk()*
- Każde się tak samo porusza: *metoda chodź()*
- Nasze klasy w żaden sposób nie są ze sobą powiązane, musimy traktować je oddzielnie, co jest dość kłopotliwe
- Nie możemy na problem spojrzeć abstrakcyjnie
- Widzimy, że przydał by się mechanizm, który pozwoli nam traktować zwierzątka na farmie *abstrakcyjnie**

*W trakcie przygotowywania wykładu nie zostało skrzywdzone żadne zwierze.

Dziedziczenie

Wstęp

- Kolejne klasy rozszerzają możliwości już istniejących
- Pozwala tworzyć nowe klasy w oparciu o już istniejące
  - Współdzielenie kodu (nasza metoda chodz())
  - Łatwiejsze zarządzanie
- Umożliwia hierarchię klas, np.: kształty $\supsetneq$ prostokąty $\supsetneq$ kwadraty
- Klasę po której dziedziczymy nazywamy **bazową**
- Klasę dziedziczącą nazywamy **pochodną**
- Obiekt klasy pochodnej, jest jednocześnie obiektem klasy bazowej, ale nie odwrotnie
- albo obiekt klasy bazowej jest subobiektem klasy pochodnej (Pochodna ma w sobie bazowy)

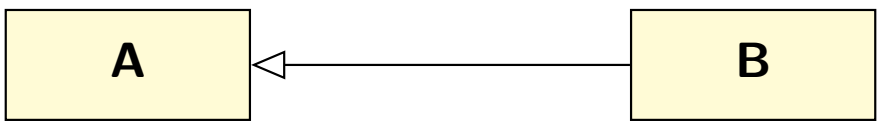
Składnia

```
class nazwa :[operator_widocznosci] nazwa_klasy_bazowej //na razie
{
  //definicja_klasy
};
```

Operator widoczności może być: *public*, ***protected***, *private*

B dziedziczy po **A**

```
class A {};
class B :public A {}; //operator public
```



Składnia

protected

Operator widoczności ***protected*** powoduje, że atrybuty klasy bazowej, mogą być swobodnie używane w klasie dziedziczącej ale poza klasą nią i klasą bazową nie są widoczne (jak *private*).

```
class A{
  public:
    int a;
  protected:
    int b;
  private:
    int c;
};
class B: public A{
  public:
  void fun(){a=8; b=4; c=5;} // c jest private!
};
int main(){
  B b;
  b.fun();
  cout << b.a << "␣" << b.b << "␣" << b.c << endl;
}
```

Składnia

Operator widoczności

- *public* → bez zmian
  - *public* → *public*
  - *protected* → *protected*
  - *private* → brak dostępu
- *protected* → *public* zmienia się w *protected*
  - *public* → ***protected***
  - *protected* → *protected*
  - *private* → brak dostępu
- *private* → wszystko zmienia się w *private*
  - *public* → ***private***
  - *protected* → ***private***
  - *private* →brak dostępu
- brak operatora → domyślnie *private*

sizeof
Tablice

```
class A{
  public:
  char tabA[1024];
  int a;
};
class B : public A{
  public:
  char tabB[1024];
  int b;
};
...
A a;
B b;
cout<<sizeof(a)<<"␣"<<sizeof(b)<<
    endl;
A tab[2];
tab[0] = B();
tab[1] = A(); //zadziala, ale ...
...
tab[0].a = 5;
tab[0].b = 5; //?
tab[1].a = 6;
```

- Obiekt klasy pochodnej zawiera w sobie subobiekt klasy bazowej
- Jaka jest odległość między tab[0] i tab[1] ?
- Czy zmieści sie tam obiekt klasy B?
- Należy uzywać tablicy wskaźników!

Dostęp do atrybutów

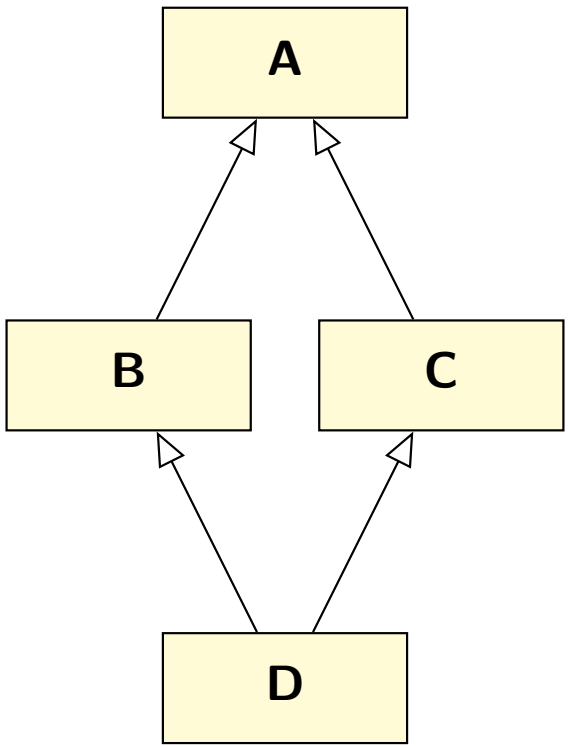
```
class Bydle{
  public:
  void jedz();
};
class kura : public bydle{
  public:
  void gdacz();
};
class krowa : public bydle{
  public:
  void mucz();
};
...
kura k; //instancja
k.jedz()
k.gdacz()

krowa * pk = new krowa();
pk->jedz(); // pointer
pk->mucz();

bydle* pb = pk;//rzutowanie
pk->jedz();
pb = &k;
k->jedz();
((bydle)k).jedz();
```

- Krowa muczy, kura gdzcze
- Kura i krowa mogą jeść
- przykład ...

```
bydle* tab[2];
tab[0] = new kura();
tab[1] = new krowa();
tab[0]->jedz();
tab[1]->jedz();
...
```

Wielodziedziczenie

Diament

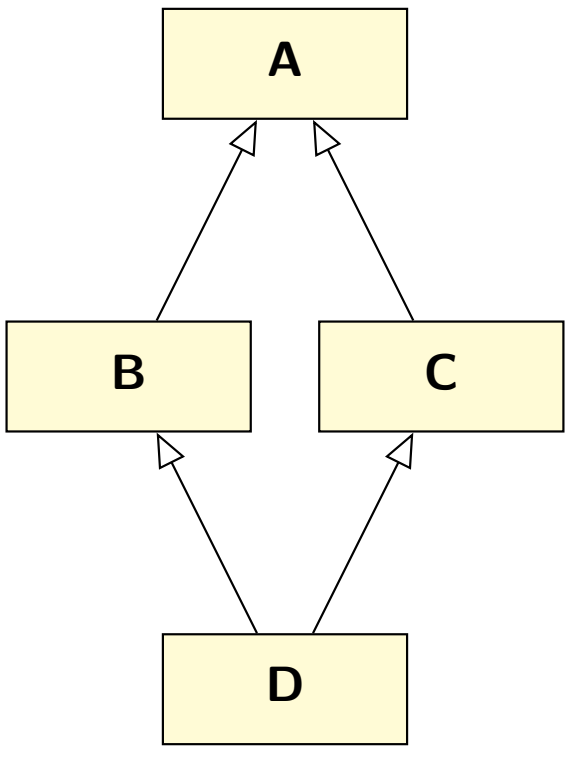


- Język C++ umożliwia wielodziedziczenie
- Może to powodować pewne komplikacje - diament
- Jak D zawiera A?
- Jaki jest rozmiar D?
- Jak jednoznacznie dostać się do atrybutów A
- Dziedziczenie wirtualne

```
class A {
    char buff[1024];
public:
    void show() {}
};
class B: public A {};
class C: public A {};
class D: public A, public B {};
```

Wielodziedziczenie

Dziedziczenie wirtualne




```
class A {
    char buff[1024];
public:
    void show() {}
};
class B: public virtual A {};
class C: public virtual A {};
class D: public A, public B {};
```

- D w jednoznaczny sposób zawiera A
- D ma odpowiedni rozmiar
- Odwołania do atrybutów A są jednoznaczne

Przykrywanie metod

```
class Bydle{
  public:
  void jedz();
  void dzwiek(){/*?*/}
};
class kura : public bydle{
  public:
  void dzwiek(){cout<<"KoKoKo"<<
       endl;}
//  void gdacz();
};
class krowa : public bydle{
  public:
  void dzwiek(){cout<<"Muuu"<<
       endl;}
//  void mucz();
};
...
kura k; k.dzwiek()
krowa a; a.dzwiek;
bydle *p = new krowa();
p->dzwiek(); //??
```

```
bydle* tab[2];
tab[0] = new kura();
tab[1] = new krowa();
tab[0]->dzwiek();
tab[1]->dzwiek();
...
```

- Zaczynamy unifikować nasze podejście
- Nie interesuje nas czym dokładnie jest bydle
- Co jest wywołane, kiedy i dlaczego?
- Przykład

Funkcje wirtualne

```
class Bydle{
  public:
  void jedz();
  virtual void dzwiek(){/*?*/}
};
class kura : public bydle{
  public:
  virtual void dzwiek()
  {cout<<"KoKoKo"<<endl;}
//  void gdacz();
};
class krowa : public bydle{
  public:
  virtual void dzwiek()
  {cout<<"Muuu"<<endl;}
//  void mucz();
};
...
kura k; k.dzwiek()
krowa a; a.dzwiek;
bydle *p = new krowa();
p->dzwiek(); //??
```

```
bydle* tab[2];
tab[0] = new kura();
tab[1] = new krowa();
tab[0]->dzwiek();
tab[1]->dzwiek();
...
```

- Pozwala na wywołanie prawidłowej metody
- tzn. metody klasy pochodnej z poziomu bazowej
- Funkcja nie jest dowiązana w czasie kompilacji, a w momencie wykonania programu - *late linking*
- Koszt pamięciowy i czasowy ... nie jeżeli dostajemy się rzutując na obiekt klasy pochodnej

Konstrukto Destruktor

Kolejność wywoływania

```
class A{
  public:
  A(){cout<<"A"<<endl;}
  A(int a){...}
  virtual ~A(){}
};
class B : public A{
  public:
  B() : A() {cout<<"B"<<endl;}
  B(int a) : A(a) {...}
  virtual ~B(){}
};
...
A a, a2(9);
B b, b1(2);
...
```

- Subobiekt klasy bazowej musi istnieć zanim powstanie obiekt dziedziczący
- Konstruktor obiektu klasy bazowej musi zostać wywołany w liście inicjalizacyjnej
- Destruktory powinny być wirtualne, by zapewnić poprawne zwalnianie pamięci
- Częste pytanie na rozmowach o pracę: Po co destruktor jest wirtualny?
- Konstruktor wywoływany jest od bazowej w dół
- Destruktor od pochodnej w górę (chyba, że nie jest wirtualny)

Metoda abstrakcyjna

```
class A{
  public:
  virtual void fun() = 0;
};
class B : public A{
  public:
  virtual void fun(){}
};
```

- Metoda nie musi posiadać ciała w klasie bazowej
- ...bo możemy nie wiedzieć jak ją zrealizować
- ...stosunek ovwodu do pola w kształcie?
- deklaracja $= 0$;
- Można wywołać taką metodę w klasie bazowej!
- Klasa posiadająca przynajmniej jedna metodę *abstrakcyjną* staje się klasą *abstrakcyjną* i nie można stworzyć już obiektu tej klasy!
- Trzeba dziedziczyć i implementować wszystkie metody abstrakcyjne

Uniemożliwienie dziedziczenia

- Przed C++11 kłopotliwe
- Prywatny konstruktor
- Dodać do klasy statyczną metodę tworzącą instancję
- Nieczytelne
- Dla każdego konstruktora musi byc metoda
- Kompilator zgłosi błąd dopiero przy instantacji klasy

```
class A{
private:
  A();
public:
  static A* zrobA() {return new
       A();}
};
class B: public A{};
int main(){
  // B b; //konstruktor prywatny
  A* pa = A::zrobA();
  delete pa;
  return 0;
}
```

Uniemożliwienie dziedziczenia

- C++11 wprowadza słówko *final*
- jak w Java?

```
class A final{
public:
  A();
};
class B: public A{};
int main(){

}
```

Lepsza farma

Przykład

Zakończymy przykładem