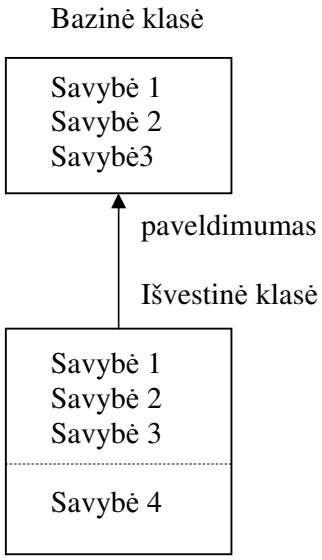
6-7-8 PASKAITOS

Turinys: Paveldimumas
Bendros žinios.
Išvestinės klasės konstruktoriai. Paveldimumas ir metodų perkrovimas.
Įvadas į abstrakčias klases.
Bendrasis ir dalinis paveldimumas.
Daugybinis paveldimumas. Daugybinio paveldimumo neapibrėžtumai.

Bendros žinios

Tai – vienas iš pagrindinių OOP principų, faktiškai leidžiantis kuriant kodą kartoti jau parašytas kodo dalis – naujų klasių kūrimas iš jau parašytų klasių. Ta jau turima klasė vadinama bazine klase, o naujosios klasės – išvestinėmis klasėmis. Išvestinė klasė paveldi visas bazinės klasės savybes ir gali jas išplėsti.
Tai – ypač patogus instrumentas taikant klasių bibliotekas: galima naudoti kitų programuotojų parašytas klases jas ne modifikuojant, o paveldint jų savybes.
UML diagrama principinei paveldimumo schemai:



Sintaksės pavyzdys: jau nagrinėta skaitiklio klasė *Counter* išplečiama išvestine klase *CounterDown*, kuri turės vieną papildomą metodą skaitiklio reikšmės mažinimui. Jei, tarkim, neturėtume prieigos prie *Counter* klasės (kad tiesiog įdėtume metodą į jos kūną), tai paveldimumas būtų pats patogiausias kelias *Counter* funkcionalumui išplėsti.

```
#include <iostream>
using namespace std;
//
class Counter{
        protected:   // 1 pastaba
```

```
        unsigned int count;
    public:
        Counter( ): count( 0 ) { }  // konstruktorius
        Counter( int c ): count( c ) { }  // konstruktorius
        unsigned int getCount( ){
            return count;
        }
        Counter operator++( ){  // "++" perkrovimas
            return Counter( ++count );
        }
};
//
class CounterDown: public Counter{   // 2 pastaba
    public:
        Counter operator--( ){
            return Counter( --count );
        }
};
//
int main( ){
    Counter c1;
    CounterDown c2;      // 3 pastaba
    cout<<"c1: "<<c1.getCount( )<<endl;
    cout<<"c2: "<<c2.getCount( )<<endl;   // 4 pastaba
    ++c2; ++c2; ++c2;                     // 4 pastaba
    cout<<"c2: "<<c2.getCount( )<<endl;
    --c2; --c2;
    cout<<"c2: "<<c2.getCount( )<<endl;
    ++c1;
    // --c1;  // 5 pastaba
    return 0;
}
```

Rezultatai:

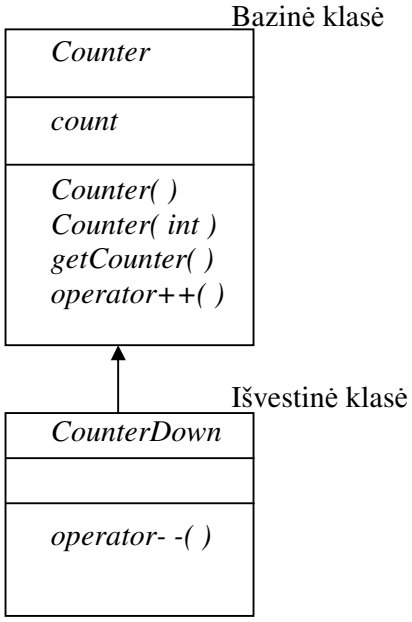
```
c1: 0
c2: 0
c2: 3
c2: 1
```

Pastabos:

1. Kad išvestinės klasės objektams būtų prieinami bazinės klasės laukai, būtinas toks raktažodis. Jei būtų *private* – jie būtų prieinami tik pačiai klasei.
2. Taip nurodoma, kad klasė paveldi kitos klasės savybes; čia *public* reiškia, kad realizuojamas bendrasis paveldimumas (apie tai skyriaus pabaigoje).
3. Išvestinės klasės konstruktorių nėra – jos objektui formuoti taikomas bazinės klasės atitinkamas konstruktorius (t. y. konstruktorius be argumentų).
4. Išvestinės klasės objektams prieinami bazinės klasės metodai *getCount( )* ir *operator++( )*, o metodui *operator--( )* - ir bazinės klasės konstruktorius su argumentais.

5. Bazinė klasė ją išplėtus nepakinta, todėl jos objektui neprieinami išvestinės klasės papildomi metodai (čia – *operator--( )*).

UML diagrama:



Išvestinės klasės konstruktoriai

Jei išvestinę ankstesnio pavyzdžio klasę reikėtų inicializuoti nenuline reikšme, operatorius

```
CounterDown c3( 10 );
```

būtų klaidingas: kompiliatorius gali taikyti tik bazinės klasės konstruktorių be argumentų. Todėl išvestinėje klasėje reikėtų įrašyti konstruktorių:

```
#include <iostream>
using namespace std;
//
class Counter{
        protected:
                unsigned int count;
        public:
                Counter( ): count( 0 ) { }
                Counter( int c ): count( c ) { }
                unsigned int getCount( ){
                        return count;
                }
                Counter operator++( ){
```

```
                return Counter( ++count );
            }
};
//
class CounterDown: public Counter{
    public:
        CounterDown( ): Counter( ){ }  // 1 pastaba
        CounterDown( int c ): Counter(  c ) { }  // 2 pastaba
        CounterDown operator--( ){
            return CounterDown( --count );   // 3 pastaba
        }
};
//
int main( ){
    CounterDown c1;
    CounterDown c2( 10 );
    cout<<"c1: "<<c1.getCount( )<<endl;
    cout<<"c2: "<<c2.getCount( )<<endl;
    ++c2; ++c2;
    CounterDown c3 = --c2;   // 4 pastaba
    cout<<"c3: "<<c3.getCount( )<<endl;
    return 0;
}
```

Pastabos:

1. Išvestinės klasės konstruktorius tokiu būdu gali kviesti bazinės klasės konstruktorių ir dar atlikti papildomus veiksmus (juos įdėtume į figūrinius skliaustus).
2. Taip argumentas perduodamas bazinės klasės konstruktoriui.
3. Dabar metodas reikšmę turi grąžinti jau naudodamasis išvestinės klasės konstruktoriumi.
4. Tokia sintaksė irgi kviečia išvestinės klasės konstruktorių su argumentu. Yra dar viena alternatyva: *CounterDown c3 (--c2);* .

Paveldimumas ir metodų perkrovimas

- kai išvestinės klasės metodų vardai sutampa su bazinės klasės metodų vardais. Kurie metodai bus kviečiami bazinės ir išvestinės klasių objektams?
Pavyzdys: steko iš trijų elementų klasė *Stack* (ji nesaugi – netikrina, ar stekas neperpildytas – tuščias) ir ją plečianti saugaus steko klasė *SafeStack*.

```
#include <iostream>
#include <cstdlib>
using namespace std;
//
class Stack{
    protected:
        enum{ MAX = 3 };
        int stack[ MAX ];
```

```
        int top;
    public:
        Stack( ): top(-1) { }
        void push( int var ) { stack[ ++top ] = var; }
        int pop( ) {return stack[ top-- ]; }
};
//
class SafeStack: public Stack {
    public:
        void push( int var ){
            if( top >= MAX-1 ){
                cout<<"Stack full"<<endl;
                exit( 1 );
            }
                Stack::push( var );  // 1 pastaba
        }
        int pop( ){
            if( top < 0 ){
                cout<<"Stack empty"<<endl;
                exit( 1 );
            }
                return Stack::pop( );  // 1 pastaba
        }

};
//
int main( ){
    SafeStack s;
    s.push( 11 );  // 2 pastaba
    s.push( 22 );
    s.push( 33 );
        cout<<s.pop( )<<endl;
        cout<<s.pop( )<<endl;
        cout<<s.pop( )<<endl;
        cout<<s.pop( )<<endl;
    return 0;
}
```

Rezultatai:

```
33
22
11
Stack empty
```

Pastaba:

1. Tokia sintaksė aiškiai parodo, kurios klasės metodas kviečiamas. Jei čia nenaudoti priklausomybės operacijos, būtų rekursiškai kviečiamas *SafeStack* atitinkamas metodas.
2. Čia kviečiamas išvestinės klasės metodas. O jei objektas *s* būtų bazinės klasės objektas – būtų kviečiamas bazinės klasės metodas.

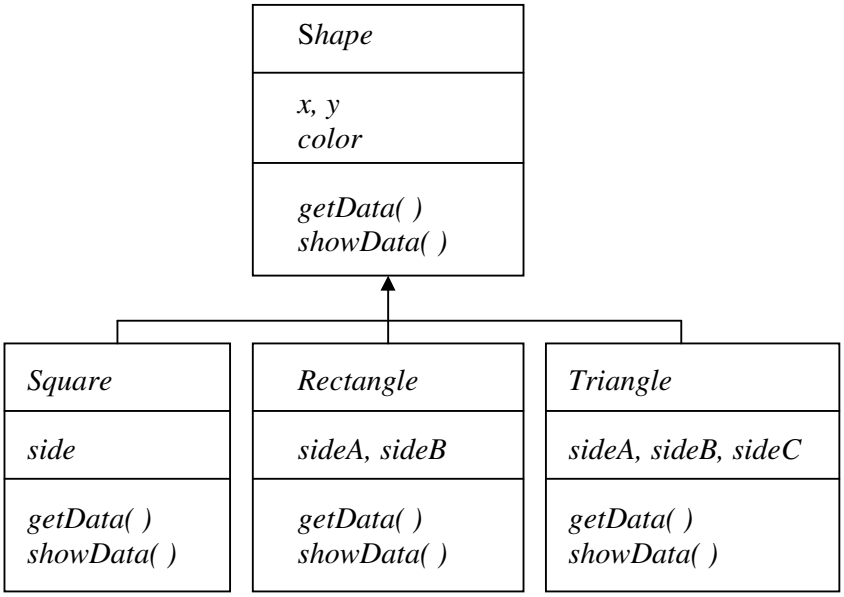
Įvadas į abstrakčias klases

Chrestomatinis paveldimumo ir polimorfizmo pavyzdys: klasė *Shape* (figūra) ir jos išvestinės klasės *Square, Rectangle* ir *Triangle*. Tegu *Shape* turi tik laukus: centrinio taško koordinates bei figūros spalvą. Kiti duomenys sutalpinami išvestinėse klasėse: kvadratui dar reikia kraštinės, stačiakampiui – abiejų kraštinių, o trikampiui – trijų kraštinių ilgių.

```
#include <iostream>
using namespace std;
//
class Shape{
        protected:
                float x, y;
                int color;
        public:
                void getData( ){
                        cout<<"Enter x, y, color: ";
                        cin>>x>>y>>color;
                }
                void showData( ){
                        cout<<"Center: "<<x<<"  "<<y<<endl;
                        cout<<"Color: "<<color<<endl;
                }
};
//
class Square: public Shape{
        private:
                float side;
        public:
                void getData( ){
                        Shape::getData( );
                        cout<<"Enter side: ";
                        cin>>side;
                }
                void showData( ){
                        Shape::showData( ):
                        cout<<"Side: "<<side<<endl;
                }
};
//
class Rectangle: public Shape{
        private:
                float sideA, sideB;
        public:
                void getData( ){
                        Shape::getData( );
                        cout<<"Enter sides: ";
                        cin>>sideA>>sideB;
```

```
                }
                void showData( ){
                        Shape::showData( ):
                        cout<<"Sides: "<<sideA<<"  "<<sideB<<endl;
                }
};
//
class Triangle: public Shape{
        private:
                float sideA, sideB, sideC;
        public:
                void getData( ){
                        Shape::getData( );
                        cout<<"Enter sides: ";
                        cin>>sideA>>sideB>>sideC;
                }
                void showData( ){
                        Shape::showData( ):
                        cout<<"Sides: "<<sideA<<"  "<<sideB<<
                                                        "  "<<sideC<<endl;
                }
};
//
int main( ){
        Square s1;
        Rectangle r1, r2;
        Triangle t1;
                s1.getData( );
                r1.getData( );
                r2 = r1;
                t1.getData( );
                        s1.showData( );
                        r2.showData( );
                        t1.showData( );
                                return 0;
}
```

Taigi klasės *Shape* objektų nėra ir jų visai nereikia – klasės tikslas yra būti tik bazine klase kitoms figūroms. Tokios klasės vadinamos abstrakčiomis (jei tiksliau, jose dar turi būti virtualūs metodai; apie tai vėliau).
UML diagrama:



Bendrasis ir dalinis paveldimumas

Šie terminai nurodo, kokią prieigą prie bazinės klasės laukų turi išvestinių klasių objektai. Bendrasis paveldimumas žymimas raktažodžiu *public*, o dalinis – *private*. Visi galimi prieigos lygiai parodyti programoje-schemoje:

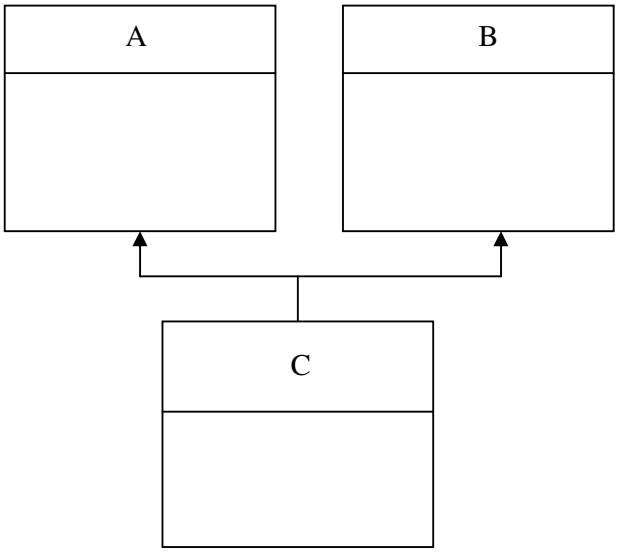
```
#include <iostream>
using namespace std;
//
class A{
        private:
                int privateData;
        protected:
                int protectedData;
        public:
                int publicData;
};
//
class B: public A{   // bendrasis paveldimumas
        public:
                void m( ){
                        int data;
                        data = privateData;   // klaida
                        data = protectedData;
                        data = publicData;
                }
};
//
```

```
class C: private A{    // dalinis paveldimumas
        public:
                void m( ){
                        int data;
                        data = privateData;   // klaida
                        data = protectedData;
                        data = publicData;
                }
};
//
int main( ){
        int data;
        B b;
        C c;
                data = b.privateData;// klaida
                data = b.protectedData;  //klaida
                data = b.publicData;
                        data = c.privateData; // klaida
                        data = c.protectedData;  //klaida
                        data = c.publicData;   // klaida
                                return 0;
}
```

Taigi abiejų paveldimumo tipų skirtumas – kad esant daliniam paveldimumui išvestinės klasės objektams neprieinami bazinės klasės jokie laukai ir metodai (paskutinis dalykas schemoje nėra iliustruotas). Dalinį paveldėjimą reikėtų taikyti, norint nuo išvestinės klasės objektų paslėpti visus bazinių klasių metodus – tai būtų aktualu, pavyzdžiui, *SafeStack* klasės atveju.

Daugybinis paveldimumas

- kai paveldima iš kelių klasių. Pavyzdys:



Sintaksė:

```
class A{ };
class B{ };
class C: public A, public B { };
```

Jei nėra bazinių klasių nesuderinamumų, tai daugybinis paveldimumas nuo vienatinio skiriasi faktiškai tik konstruktorių naudojimu.
Pavyzdys. Rašoma klasė *Lumber* informacijai apie medienos gaminius talpinti. Tarkim, ji gali dalį funkcionalumo paveldėti iš jau parašytos klasės *Distance* (*int* ir *float* formato laukus - gaminio ilgį pėdomis ir coliais) ir dalį – iš klasės *Type* (*string* formato – skerspjūvio duomenis ir rūšį), o šiuos laukus dar papildo *int* formato duomeniu kiekiui ir *double* – kainai talpinti.

```
#include <iostream>
#include <string>
using namespace std;
//
class Type{
        protected:
                string dimensions;
                string grade;
        public:
                Type( ): dimensions( "N/A" ), grade( "N/A" ) { }
                Type( string d, string g ): dimensions( d ), grade( g ) { }
                void getType( ){
                        cout<<"Enter dimensions and grade:";
                        cin>>dimensions>>grade;
                }
                void showType( ){
                        cout<<"Dimensions: "<<dimensions<<" Grade: "<<grade
                                                                <<endl;
                }
};
//
class Distance{
        private:
                int feet;
                float inches;
        public:
                Distance( ): feet( 0 ), inches( 0.f ) { }
                Distance( int ft, float in): feet( ft ), inches( in ) { }
                void getDistance( ){
                        cout<<"Enter feet and inches:";
                        cin>>feet>>inches;
                }
                void showDistance( ){
                        cout<<"Length: "<<feet<<" - "<<inches<<endl;
                }
};
```

```
//
class Lumber: public Type, public Distance{
        private:
                int quantity;
                double price;
        public:
                Lumber( ): Type( ), Distance( ), quantity( 0 ), price( 0. ) { }      // 1
                Lumber( string d, string g, int ft, float in, int q, double p ):
                        Type( d, g ), Distance( ft, in ), quantity ( q ), price( p ) { }  // 2
                void getLumber( ){
                        Type::getType( );       // 3
                        Distance::getDistance( );
                        cout<<"Enter quantity and price:";
                        cin>>quantity>>price;
                }
                void showLumber( ){
                        Type::showType( );
                        Distance::showDistance( );
                        cout<<"Quantity and price: "<<quantity<<"  "<<price
                                                                <<endl;
                }
};
//
int main( ){
        Lumber p1;
        p1.getLumber( );
        Lumber p2( "2x4", "A", 8, 0.f, 100, 50.00 );
        p1.showLumber( );
        p2.showLumber( ;
        return 0;
}
```

Ši programa neatlieka jokių veiksmų, tik įveda dviejų *Lumber* klasės objektų duomenis ir juos parodo.
Pastabos:

1. Taip kviečiami abiejų bazinių klasių konstruktoriai, o vėliau užpildomi išvestinės klasės papildomi laukai.
2. Tokia sintaksė konstruktoriams su argumentais.
3. Bazinės klasės metodo kvietimui būtina priklausomybės operacija "::".

Daugybinio paveldimumo neapibrėžtumai

1.

```
#include <iostream>
using namespace std;
//
class A{
        public:
```

```
                void m( ){ }
};
//
class B{
        public:
                void m( ){ }
};
//
class C: public A, public B{ };
//
int main( ){
        C c;
        c.m( );     //  klaida! Neaišku, kurį metodą kviesti
        c.A::m( );
        c.B::m( );
        return 0;
}
```

2.

```
#include <iostream>
using namespace std;
//
class A{
        public:
                void m( ){ }
};
//
class A1: public A{ };
//
class A2: public A{ };
//
class B: public A1, public A2{ };
//
int main( ){
        B b;
        b.m( );    // klaida! Neaišku, kurį metodą kviesti:
                   // A1 ir A2 turi m( ) kopijas, paveldėtas iš A
        return 0;
}
```