Język C++

część 7

polimorfizm i funkcje wirtualne
klasy abstrakcyjne

Jarosław Gramacki
Instytut Informatyki i Elektroniki

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne
  - UWAGA! Wymagana znajomość tematyki wykładu nr **6**
  - cytat z poprzedniego wykładu:
    funkcja `oblicz_place()` w klasie bazowej jest tylko "oparciem" dla słowa `virtual`. Inaczej pojawi się błąd przy kompilacji linii
    `total += table[ i ]->oblicz_place();`
  - w rzeczywistości funkcji `Personel::oblicz_place()` prawdopodobnie nigdy nie wywołamy. Skorzystamy z jej wersji przedefiniowanych w klasie (klasach) pochodnej
  - trudno nawet sobie wyobrazić rzeczywisty obiekt typu `Personel` !, gdyż jest on zbyt ogólny do opisu konkretnego pracownika
  - w takich wypadkach powinno się deklarować tzw. czyste funkcje wirtualne (ang. *pure virtual function*)

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne

  – Niekiedy definiujemy klasę reprezentującą jakąś abstrakcyjna koncepcję, opisującą pewne własności wspólne dla reprezentowanej abstrakcji.

  – Przykładem takiej koncepcji abstrakcyjnej niech będzie pojecie: ”figura”. Tworzenie obiektu klasy figura nie ma sensu, ze względu na jego abstrakcyjność, ale już podklasy figury, np. kwadrat, trójkąt czy koło mogą być instancjowane.

  – Klasa abstrakcyjna może zawierać metody abstrakcyjne, które nie zawierają implementacji. W ten sposób klasa abstrakcyjna definiuje interfejs programistyczny, który musi znaleźć się w nie-abstrakcyjnych jej podklasach.

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne

  – klasę, która zawiera co najmniej jedną czystą funkcję wirtualną nazywamy klasą abstrakcyjną. Język C++ nie pozwala na utworzenie obiektu takiej klasy ! (obiekt typu `Personel` w naszym przypadku nie ma sensu !)

  – nie oznacza to, że nie można utworzyć wskaźnika do takiej klasy. Wskaźnik taki ma jak najbardziej sens - patrz np. klasa `TablePersonel`

  – przyrostek "`=0`" jawnie mówi kompilatorowi, że funkcja ta NIE będzie implementowana w bieżącej klasie. Jest to funkcja z tzw. zerowym zbiorem instrukcji (w odróżnieniu od funkcji z pustym zbiorem instrukcji {})

```
class Personel {
public:
    Personel( const char* _nazwisko, short _biuro = 0 );
    ~Personel() { delete [] nazwisko; }
    virtual void print() const;
    // virtual float oblicz_place() const { return 0; } // tak było
    virtual float oblicz_place() const =0;          // tak jest teraz:
                                                    // czysta funkcja wirtualna
    void set_biuro( short _biuro ) { biuro = _biuro; }
    const char* get_telefon() const;
private:
    char* nazwisko;
    short biuro;
};
```

dla czytelności bez spacji pomiędzy = oraz 0

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne
  - czyste funkcje wirtualne i klasy abstrakcyjne stosowane są do tworzenia bardzo ogólnego interfejsu ukierunkowanego na cała hierarchię klas
  - wtedy jedynie klasy pochodne reprezentują rzeczywiste obiekty
  - polimorfizm jest mechanizmem mocno" obciążającym" program. Stąd można go „wyłączyć" poprzez niestosowanie funkcji wirtualnych (wtedy program działa szybciej). W języku Java nie ma takiej możliwości.
  - w Javie wszystkie metody zachowują się jak metody wirtualne w C++.
  - w Javie wprowadzono tzw. interfejsy
  - Jest to jednak temat na zupełnie inny wykład

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne
  - ogólny przykład klasy abstrakcyjnej - szkielet

```
class Ksztalt     // klasa abstrakcyjna
{
public:
    virtual void rysuj() =0;
    virtual void obracaj (float _kat) =0;
    virtual void przesun (float _x, float _y) =0;
    ... };

class Okrag : public Ksztalt {
public:
    void rysuj();
    void obracaj (float _kat);
    void przesun (float _x, float _y);
private:
    float srodek, promien;  };

class Prostokat : public Ksztalt {
public:
    void rysuj();
    void obracaj (float _kat);
    void przesun (float _x, float _y);
private:
    float gorny, lewy, dolny, prawy; };

class Kwadrat : public Prostokat {...};
```

Klasa `Ksztalt` definiuje ogólny interfejs, którego będziemy potrzebowali do przetwarzania kształtu geometrycznego

Tu znajdują się rzeczywiste implementacje czystych funkcji wirtualnych klasy `Ksztalt`

- Funkcje wirtualne i polimorfizm - przykład "bazowy"

```
#include <iostream>    using namespace std;

class CPolygon {
  protected:     int width, height;
  public:        void set_values (int a, int b)  { width = a; height = b; }
  };

class CRectangle: public CPolygon {
  public:
    int area () { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()  { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle  trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << endl;     // 20
  cout << trgl.area() << endl;     // 10
  return 0;   }
```

wskaźniki do klasy bazowej

- Funkcje wirtualne i polimorfizm - przykład "bazowy"

```
#include <iostream> using namespace std;

class CPolygon {
  protected:     int width, height;
  public:        void set_values (int a, int b)  { width=a; height=b; }
                 virtual int area ()             { return (0); }
  };

class CRectangle: public CPolygon {
  public:    int area () { return (width * height); }   };

class CTriangle: public CPolygon {
  public:    int area ()  { return (width * height / 2); }   };

int main () {
  CRectangle rect;   CTriangle trgl;   CPolygon poly;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  CPolygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << endl;  // 20
  cout << ppoly2->area() << endl;  // 10
  cout << ppoly3->area() << endl;  // 0
  return 0;    }
```

po usunięciu `virtual` z `CPolygon` tu pojawią się zera

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne
  - przykład klasy abstrakcyjnej - kompletny przykład

```
// shape.h
// Shape abstract-base-class definition.

#ifndef SHAPE_H
#define SHAPE_H

#include <string>                  // C++ standard string class
using std::string;
class Shape {

public:

   // virtual function that returns shape area
   virtual double getArea() const;

   // virtual function that returns shape volume
   virtual double getVolume() const;

   // pure virtual functions; overridden in derived classes
   virtual string getName() const =0;              // return shape name
   virtual void print() const =0;                  // output shape

};

#endif
```

stosuj zawsze dyrektywy dla preprocesora uniemożliwiające wielokrotne analizowanie tych samych informacji

virtual and pure virtual functions.

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne
  - przykład klasy abstrakcyjnej - kompletny przykład

```
// shape.cpp
// Shape class member-function definitions.

#include <iostream>
using std::cout;
#include "shape.h"                          // Shape class definition


// return area of shape; 0.0 by default
double Shape::getArea() const
{
   return 0.0;
}


// return volume of shape; 0.0 by default
double Shape::getVolume() const
{
   return 0.0;
}
```

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne

```
// point.h
// Definition of class Point

#ifndef POINT_H
#define POINT_H
#include "shape.h"  // Shape class definition

class Point : public Shape {

public:
   Point( int = 0, int = 0 );      // default constructor

   void setX( int );               // set x in coordinate pair
   int getX() const;               // return x from coordinate pair

   void setY( int );               // set y in coordinate pair
   int getY() const;               // return y from coordinate pair

   // return name of shape (i.e., "Point" )
   virtual string getName() const;

   virtual void print() const;     // output Point object

private:
   int x; int y;
};
#endif
```


- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne
  - przykład klasy abstrakcyjnej - kompletny przykład

```
// point.cpp
// Member function definitions for class Point

#include <iostream>
using std::cout;
#include "point.h"   // Point class definition

Point::Point( int xValue, int yValue )
             : x( xValue ), y( yValue )
{ }

void Point::setX( int xValue ) {   x = xValue; }
int Point::getX() const        {   return x; }
void Point::setY( int yValue ) {   y = yValue; }
int Point::getY() const        {   return y; }

// override pure virtual function getName: return name of Point
string Point::getName() const {
   return "Point";
}

// override pure virtual function print: output Point object
void Point::print() const {
   cout << '[' << getX() << ", " << getY() << ']';
}
```

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne

```
// circle.h
// Definition of class Circle

#ifndef CIRCLE_H
#define CIRCLE_H
#include "point.h"                          // Point class definition

class Circle : public Point {

public:
   Circle( int = 0, int = 0, double = 0.0 );
   void setRadius( double );                // set radius
   double getRadius() const;                // return radius

   double getDiameter() const;              // return diameter
   double getCircumference() const;         // return circumference
   virtual double getArea() const;          // return area

   // return name of shape (i.e., "Circle")
   virtual string getName() const;

   virtual void print() const;              // output Circle object

private:
   double radius;                           // Circle's radius
};
#endif
```


- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne

```
// circle.cpp
// Member function definitions for class Circle

#include <iostream>
using std::cout;
#include "circle.h"                    // Circle class definition

Circle::Circle( int xValue, int yValue, double radiusValue )
               : Point( xValue, yValue )
{ setRadius( radiusValue ); }

void Circle::setRadius( double radiusValue ) {
radius = ( radiusValue < 0.0 ? 0.0 : radiusValue ); }
double Circle::getRadius() const        { return radius; }

// calculate and return diameter
double Circle::getDiameter() const      { return 2 * getRadius(); }

// calculate and return circumference
double Circle::getCircumference() const { return 3.14159 * getDiameter(); }

// override virtual function getArea: return area of Circle
double Circle::getArea() const { return 3.14159 * getRadius() * getRadius(); }

// override pure virutual function getName: return name of Circle
string Circle::getName() const { return "Circle"; }

// override pure virtual function print: output Circle object
void Circle::print() const {
   cout << "center is ";
   Point::print();                      // invoke Point's print function
   cout << "; radius is " << getRadius();  }
```

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne

```
// cylindr.h
// Definition of class Cylinder

#ifndef CYLINDR_H
#define CYLINDR_H
#include "circle.h"  // Circle class definition

class Cylinder : public Circle {

public:
   Cylinder( int = 0, int = 0, double = 0.0, double = 0.0 );

   void setHeight( double );                // set Cylinder's height
   double getHeight() const;                // return Cylinder's height

   virtual double getArea() const;          // return Cylinder's area
   virtual double getVolume() const;        // return Cylinder's volume

   // return name of shape (i.e., "Cylinder" )
   virtual string getName() const;

   virtual void print() const;              // output Cylinder

private:
   double height;                           // Cylinder's height
};
#endif
```


- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne

```
// cylinder.cpp
// Member and friend function definitions for class Cylinder

#include <iostream>   using std::cout;   #include "cylinder.h"

Cylinder::Cylinder( int xValue, int yValue, double radiusValue,  double heightValue )
  : Circle( xValue, yValue, radiusValue ) { setHeight( heightValue ); }

// set Cylinder's height
void Cylinder::setHeight( double heightValue ) {
   height = ( heightValue < 0.0 ? 0.0 : heightValue ); }

// get Cylinder's height
double Cylinder::getHeight() const  { return height; }

// override virtual function getArea: return Cylinder area
double Cylinder::getArea() const {
   return 2 * Circle::getArea() +  getCircumference() * getHeight(); }

// override virtual function getVolume: return Cylinder volume
double Cylinder::getVolume() const { return Circle::getArea() * getHeight(); }

// override pure virtual function getName: return name of Cylinder
string Cylinder::getName() const { return "Cylinder"; }

// override pure virtual function print: output Cylinder object
void Cylinder::print() const {
   Circle::print();
   cout << "; height is " << getHeight(); }
```

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne

```
// main.cpp (Driver for shape, point, circle, cylinder hierarchy)

#include <iostream.h> #include <iomanip.h>  #include "shape.h"
#include "point1.h"   #include "circle1.h"  #include "cylindr1.h"

void virtualViaPointer ( const Shape * );
void virtualViaReference ( const Shape & );

int main() {
   cout << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 );

   Point point( 7, 11 );                  // create a Point
   Circle circle( 3.5, 22, 8 );           // create a Circle
   Cylinder cylinder( 10, 3.3, 10, 10 );  // create a Cylinder

   point.printShapeName();    // static binding
   point.print();             // static binding
   cout << '\n';

   circle.printShapeName();   // static binding
   circle.print();            // static binding
   cout << '\n';

   cylinder.printShapeName(); // static binding
   cylinder.print();          // static binding
   cout << "\n\n";
```


- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne

```
   Shape *arrayOfShapes[ 3 ];  // array of base-class pointers

   // aim arrayOfShapes[0] at derived-class Point object
   arrayOfShapes[ 0 ] = &point;

   // aim arrayOfShapes[1] at derived-class Circle object
   arrayOfShapes[ 1 ] = &circle;

   // aim arrayOfShapes[2] at derived-class Cylinder object
   arrayOfShapes[ 2 ] = &cylinder;

   // Loop through arrayOfShapes and call virtualViaPointer
   // to print the shape name, attributes, area, and volume
   // of each object using dynamic binding.
   cout << "Virtual function calls made off "
        << "base-class pointers\n";

   for ( int i = 0; i < 3; i++ )  virtualViaPointer( arrayOfShapes[ i ] );

   // Loop through arrayOfShapes and call virtualViaReference
   // to print the shape name, attributes, area, and volume
   // of each object using dynamic binding.
   cout << "Virtual function calls made off "
        << "base-class references\n";

   for ( int j = 0; j < 3; j++ ) virtualViaReference( *arrayOfShapes[ j ] );
   return 0;    }
```

- Funkcje wirtualne i polimorfizm - klasy abstrakcyjne

```
// Make virtual function calls off a base-class pointer
// using dynamic binding.

void virtualViaPointer( const Shape *baseClassPtr )
{
   baseClassPtr -> printShapeName();
   baseClassPtr -> print();
   cout << "\nArea = " << baseClassPtr -> area()
        << "\nVolume = " << baseClassPtr -> volume() << "\n\n";
}

// Make virtual function calls off a base-class reference
// using dynamic binding.

void virtualViaReference( const Shape &baseClassRef )
{
   baseClassRef.printShapeName();
   baseClassRef.print();
   cout << "\nArea = " << baseClassRef.area()
        << "\nVolume = " << baseClassRef.volume() << "\n\n";
}
```


- Funkcje wirtualne i polimorfizm - przykład "bazowy"

```
#include <iostream>     using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b) { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void)  { cout << this->area() << endl; }
  };

class CRectangle: public CPolygon {
  public:     int area (void) { return (width * height); }   };

class CTriangle: public CPolygon {
  public:     int area (void)  { return (width * height / 2); }   };

int main () {
  CPolygon * ppoly1 = new CRectangle;
  CPolygon * ppoly2 = new CTriangle;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();            // 20
  ppoly2->printarea();            // 10
  delete ppoly1;
  delete ppoly2;
  return 0;    }
```

alokacja dynamiczna a polimorfizm

wskaźniki (jak poprzednio) do klasy bazowej ale obiekty typu "prostokąt" lub "trójkąt !

- Funkcje wirtualne i polimorfizm - uwagi
  - Uwagi na zakończenie
  - funkcje wirtualne są szczególnie atrakcyjne podczas przetwarzania wszystkich obiektów pewnej hierarchii za pośrednictwem wskaźnika do jej klasy bazowej (a dzięki konwersjom wskaźnik ten może wskazywać na dowolną klasę pochodną w tej samej hierarchii)
  - dzięki polimorfizmowi łatwo jest rozszerzać funkcjonalność (dobrze napisanych!) bibliotek obiektowych (nawet gdy nie posiadamy ich kodów źródłowych)
  - wady polimorfizmu: rozwiązanie bardzo "zasobochłonne". Między innymi z tego powodu zrezygnowano z polimorfizmu w bibliotece STL