Objektum orientált programozás C++ nyelven 6. Polimorfizmus

PEKÁRDY MILÁN – PANNON EGYETEM

PEKARDY@DCS.UNI-PANNON.HU

Mutatók ősosztályra

- Az öröklés egyik legfontosabb eredménye, hogy a származtatott osztályra mutató mutatók típus kompatibilisek (type-compatible) az ősosztályukra mutató mutatóval,
- Ez azt jelenti, hogy egy ősosztály típusú mutató mutathat egy származtatott osztályra (fordítva csak valamilyen konverzióval – pl.: dynamic_cast – oldható meg),
- A polimorfizmus ennek a sokoldalú funkciónak az előnyös kihasználása

Mutatók ősosztályra

- A példában láthatjuk, hogy két ősosztály mutatót hozunk létre úgy, hogy egy-egy származtatott osztályra mutatnak igazából,
- Mivel ősosztály mutatókat használunk, ezért ezeken keresztül csak az ősosztályban definiált tagok érhetők el (az area függvényt a megfelelő származtatott típusokon keresztül érjük el)
- Ha az ősosztályban is definiáljuk az area metódust, akkor az ősosztály mutatókon keresztül is elérhetnénk azt, de mivel a származtatott osztályokban más-más a megvalósítása, ezért nehézkes egy helyen implementálni

```
// pointers to base class
#include <iostream>
using namespace std;

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

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

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

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

Virtuális tagok

- Egy virtuális függvény olyan függvény, ami újra definiálható a származtatott osztályokban úgy, hogy az ősosztály referenciákon keresztül hívható marad,
- Virtuális függvényt a virtual kulcsszóval definiálunk,
- Virtuális függvények a származtatott osztályokban felüldefiniálhatók és az ősosztály mutatón keresztüli híváskor a megfelelő származtatott verzió fut le,
- Nem virtuális tagok is felüldefiniálhatók a származtatott osztályokban, de ebben az esetben az ősosztály referenciákon keresztül nem a származtatott verziót, hanem az ősosztály verziót érjük el (távolítsuk el a példában a virtual kulcsszót az area függvény elől és figyeljük meg az eredményt),
- Az olyan osztályt, ami virtuális függvényt deklarál vagy örököl polimorf osztálynak (polymorphic class) nevezzük.

```
// virtual members
#include <iostream>
using namespace std;

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

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

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

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

Absztrakt ősosztályok

- Egy absztrakt ősosztály olyan osztály, ami csak szülő osztályként használható, így tartalmazhat definíció nélküli virtuális függvényeket (tisztán virtuális függvények – pure virtual functions),
- A tisztán virtuális függvények törzsét az =0 karakterekkel helyettesítjük,
- A legalább egy tisztán virtuális függvényt tartalmazó osztályokat absztrakt ősosztályoknak nevezzük,
- Az absztrakt ősosztályok közvetlenül nem használhatók példányok létrehozására

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

```
Polygon mypolygon;   // not working if Polygon is abstract base class
```

Absztrakt ősosztályok

- Az absztrakt ősosztályokat használhatjuk arra, hogy mutatókat definiálunk velük a különböző származtatott osztályokra és így kihasználjuk a polimorf tulajdonságokat,
- A példában láthatjuk, hogy a származtatott osztályokra hivatkozhatunk a közös ősosztály mutatókkal, és ezeken a mutatókon keresztül hívhatjuk a virtuális függvényeket, amely hívások a megfelelő származtatott verziókhoz futnak be

```
// abstract base class
#include <iostream>
using namespace std;

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

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

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

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  return 0;
}
```

Absztrakt ősosztályok

- Egy absztrakt ősosztályban használhatjuk a speciális this mutatót a megfelelő virtuális tagok elérésére,
- Az objektum-orientált programozás egyik alappillére a polimorfizmus, a C++ nyelv ezt a bemutatott virtuális tagok és absztrakt ősosztályok segítségével támogatja,
- Az eddigi alappéldákon túl használhatjuk a polimorf tulajdonságokat objektumok tömbjére, dinamikusan foglalt objektumokra, stb.

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;

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

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

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

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  return 0;
}
```

Absztrakt ősosztályok

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height/2; }
};
```

```
int main () {
  Polygon * ppoly1 = new Rectangle (4,5);
  Polygon * ppoly2 = new Triangle (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

Típus konverziók

(TYPE CONVERSIONS)

Implicit konverzió (implicit conversion)

- Implicit konverzió automatikusan történik, amikor egy érték átmásolódik egy kompatibilis típusba:

```
short a=2000;
int b;
b=a;
```

- A példában egy short „előlép” (promote) int típussá explicit operátor használata nélkül (sztenderd konverzió),
- Sztenderd konverzió végezhető a szám típusok (numerical types), logikai értékek (bool) és bizonyos mutatók között,
- Amikor egy kisebb típusból konvertálunk nagyobb típusra azt „előléptetésnek” (promotion) nevezik, ilyenkor biztosított, hogy ugyanazt az értéket kapjuk a cél típusban

Implicit konverzió

- Bizonyos konverziók esetében nem biztosított kiinduló érték reprezentálása:
  - Ha egy negatív egész értéket konvertálunk egy előjel nélküli típusra, akkor az eredményt az eredeti érték kettes komplemens bitenkénti reprezentációja adja,
  - Logikai értékre/ről való konverzió esetén a 0 számérték és a null mutató érték mindig hamisat ad, minden más pedig igazat (1-es érték),
  - Ha egy lebegőpontos értéket konvertálunk egy egész típusra, akkor a tört részt levágjuk. Ha az eredmény kívül esik a cél típus tartományán, akkor a viselkedés nem meghatározott (undefined behavior),
  - Ha a konverzió azonos szám típusok között van, akkor a konverzió megengedett, de az eredmény implementáció specifikus (implementation-specific) és nem biztos, hogy „hordozható” (portable)
- Előfordulhat olyan konverzió, amikor levágunk az eredeti értékből vagy nem tudjuk olyan pontossággal ábrázolni, mint az eredeti érték, ilyenkor a fordító ezt a figyelmeztetéssel jelezheti, ez a figyelmeztetés explicit konverzióval elkerülhető

Implicit konverzió

- Nem elemi típusok esetén a tömbök és függvények implicit mutatókká konvertálódnak és a mutatók általánosan a következő konverziókat engedik:
  - Null mutatók tetszőleges típusú mutatókká konvertálhatók,
  - Tetszőleges típusra mutató mutatók void mutatókká konvertálhatók,
  - Mutató „felkonvertálás” (upcast): származtatott osztályokra mutató mutatók konvertálhatók elérhető (accessible) és egyértelmű (unambiguous) szülő osztály típusú mutatóra

Implicit konverzió osztályokkal

- Osztályok esetén az implicit konverzió három tagfüggvény segítségével kontrollálható:
  - egy-paraméteres konstruktor (single-argument constructor): segítségével lehetséges az implicit konverzió egy adott típusból az objektum inicializálásával,
  - Értékadó operátor (assignment operator): megengedi az implicit konverziót egy adott típusból értékadáskor,
  - Típus-konverziós operátor (type-cast operator): megengedi az implicit konverziót egy adott típusra. A típus-konverziós operátor speciális szintaktikával rendelkezik: az operator kulcsszót a céltípus követi majd egy üres zárójelpár.

Implicit konverzió osztályokkal

```
// implicit conversion of classes:
#include <iostream>
using namespace std;

class A {};

class B {
public:
  // conversion from A (constructor):
  B (const A& x) {}
  // conversion from A (assignment):
  B& operator= (const A& x) {return *this;}
  // conversion to A (type-cast operator)
  operator A() {return A();}
};

int main ()
{
  A foo;
  B bar = foo;     // calls constructor
  bar = foo;       // calls assignment
  foo = bar;       // calls type-cast operator
  return 0;
}
```

explicit kulcsszó

- C++-ban egy függvény hívásakor egy implicit konverzió történhet argumentumonként. A függvény hívásakor előfordulhat, hogy nem kívánatos a konverzió,
- A konverzió megelőzésére használhatjuk az explicit kulcsszót a konverziós konstruktor megjelölésével,
- Így nem hívhatjuk a függvényt a konvertálás forrástípusával megegyező objektummal, továbbá az értékadás stílusú szintaktika is tiltott,
- A konverziós konstruktoron kívül a többi típus-konverziós tag is megjelölhető az explicit kulcsszóval, így az implicit konverzió megelőzhető

```
// explicit:
#include <iostream>
using namespace std;

class A {};

class B {
public:
  explicit B (const A& x) {}
  B& operator= (const A& x) {return *this;}
  operator A() {return A();}
};

void fn (B x) {}

int main ()
{
  A foo;
  //B bar = foo; // not allowed
  B bar (foo);
  bar = foo;
  foo = bar;

//  fn (foo);  // not allowed for explicit ctor.
  fn (bar);

  return 0;
}
```

Típus kasztolás (type casting)

- Azok a konverziók, amelyek az érték egy másik interpretációjához vezetnek (pl. egy típus mutatót egy másik típus mutatóra konvertálnánk) explicit konverziót igényelnek, ezt a műveletet hívják a C++-ban típus kasztolásnak,
- Két féle szintaktikát használhatunk:

```
double x = 10.3;
int y;
y = int (x);     // functional notation
y = (int) x;     // c-like cast notation
```

Típus kasztolás

- Az alapvető típusoknál legtöbbször jól alkalmazható a kasztolás, azonban osztályokon, illetve osztályokra mutató mutatókon problémás lehet (szintaktikailag helyes a kód, de futási hiba előfordulhat):
- A példában mutatókon keresztül az explicit típus kasztolás segítségével két nem kapcsolódó osztály példányát kapcsoljuk össze. Futás közben viszont a függvényhívás hibás lesz.

```
// class type-casting
#include <iostream>
using namespace std;

class Dummy {
    double i,j;
};

class Addition {
    int x,y;
  public:
    Addition (int a, int b) { x=a; y=b; }
    int result() { return x+y;}
};

int main () {
  Dummy d;
  Addition * padd;
  padd = (Addition*) &d;
  cout << padd->result();
  return 0;
}
```

Típus kasztolás

- Az előző példából látható, hogy az ellenőrzés nélküli explicit konverzió nem várt hibákhoz vezethet,
- Az ilyen típusú konverziók megfelelő kontrollálásához a következő operátorokat használhatjuk:
  - `dynamic_cast <new_type> (expression)`
  - `reinterpret_cast <new_type> (expression)`
  - `static_cast <new_type> (expression)`
  - `const_cast <new_type> (expression)`
- A new_type a cél típus, ami egy <> párban van, utána zárójelben megadjuk a kifejezést (expression), aminek az eredményét konvertálni szeretnénk a cél típusra.

dynamic_cast

- Csak mutatókkal vagy osztály referenciákkal (void*) használható,
- Célja, hogy a konverzió eredményeként visszaadott érték egy érvényes teljes objektumra mutasson a cél típusnak megfelelően,
- Általában mutató „felkonvertálásra” (pointer upcast) használjuk (származtatott osztály mutató ősosztály mutatóra),
- De mutató „lekonvertálás” (pointer downcast) is lehetséges polimorf osztályok esetén, akkor ha a mutatott objektum egy érvényes teljes objektuma a cél típusnak (ősosztály mutató származtatott mutatóra).

dynamic_cast

- A példában az első esetben az ősosztály mutató egy származtatott osztály típusra mutat valójában így a konverzió sikeres. A második esetben viszont egy ősosztály objektumra mutat, ami nem konvertálható egy érvényes teljes származtatott típusú objektumra, ezért a konverzió sikertelen

```
// dynamic_cast
#include <iostream>
#include <exception>
using namespace std;

class Base { virtual void dummy() {} };
class Derived: public Base { int a; };

int main () {
  try {
    Base * pba = new Derived;
    Base * pbb = new Base;
    Derived * pd;

    pd = dynamic_cast<Derived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast.\n";

    pd = dynamic_cast<Derived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast.\n";

  } catch (exception& e) {cout << "Exception: " << e.what();}
  return 0;
}
```

dynamic_cast

- Amikor a konverzió azért nem sikerül, mert a mutatott érték nem egy teljes érvényes objektuma a cél típusnak, akkor az operátor null mutatóval tér vissza,
- Amikor a konverziót referencia típusra alkalmazzuk és a konverzió nem lehetséges, akkor egy `bad_cast` kivételt dob az operátor,
- Az operátor képes az implicit konverzióknál említett konverziók elvégzésére is: null mutatók konvertálása, tetszőleges típusú mutatók konvertálása `void*`-ra, stb.
- Az operátor megfelelő működéséhez szükséges a futás idejű típus információ (Run-Time Type Information – RTTI), néhány fordító opcionálisan támogatja ezt, ezért az ilyen fordítók esetén szükséges az RTTI engedélyezése

static_cast

- Konvertálni képes kapcsolódó osztályokra mutató mutatók között,
- Le- és felkonvertálást is tud (downcast/upcast),
- A konvertálásnál nincs típus ellenőrzés (konvertálandó objektum teljes objektuma a cél típusnak), így a programozó feladata a konverzió helyességének biztosítása,
- Mivel nincs ellenőrzés így nem szükséges típus információ hozzá, így gyorsabb is.

```
class Base {};
class Derived: public Base {};
Base * a = new Base;
Derived * b = static_cast<Derived*>(a);
```

static_cast

- A következő konverziókra képes:
  - void*-ból tetszőleges mutató típusra: biztosítja, hogy ha a void* ugyanolyan típusból származott mint a cél típus, akkor az eredmény is olyan típus lesz,
  - Egész, lebegőpontos és enum típusok konvertálása enum típusra,
  - Explicit hívhatja az egy paraméteres konverziós konstruktort vagy a konverziós operátort,
  - Jobb-érték referenciákra (rvalue references) konvertálni,
  - Enum osztály értékek konvertálása egész vagy lebegőpontos értékekre,
  - Tetszőleges típust void-ra konvertálni (az értéket kiértékeli majd eldobja).

reinterpret_cast

- Tetszőleges mutató típust tetszőleges másik mutató típusra konvertál (nem kapcsolódó osztályok között is),
- Az eredmény egy bináris másolat az egyik mutató által mutatott értékről a másikra,
- Minden konverzió megengedett: sem a mutatott értékre, sem a mutató típusára nem történik ellenőrzés,
- Képes mutatókat is konvertálni egész típusokra és fordítva is. A pontos reprezentáció platform függő. Az egyetlen garancia, hogy az egész típus elegendően nagy, hogy tárolni tudja a mutatót,
- A konverzió eredménye (a típusok bináris reprezentációjának „újraértelmezése”) sokszor rendszer függő így nem hordozható:
- A példa lefordul, de nincs sok értelme, mert két nem kapcsolódó osztályt kapcsol össze.

```
class A { /* ... */ };
class B { /* ... */ };
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

const_cast

- Ebben az esetben a kasztolás egy mutató által mutatott objektum konstans mivoltát módosítja: vagy beállítja vagy törli,
- Akkor lehet hasznos, ha konstans mutatót akarunk egy olyan függvénynek átadni, ami nem konstans argumentumot vár:
- A példában nincs gond a konstans kasztolásból, mert a függvény nem módosítja az értéket,
- Viszont ha eltávolítjuk a konstans tulajdonságot egy objektumról és módosítjuk is azt, akkor az nem meghatározott viselkedést okoz.

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

void print (char * str)
{
  cout << str << '\n';
}

int main () {
  const char * c = "sample text";
  print ( const_cast<char *> (c) );
  return 0;
}
```

typeid

- A typeid segítségével egy kifejezés típusát ellenőrizhetjük:
  - `typeid (expression)`
- az operátor egy `type_info` típusú konstans objektum referenciával tér vissza,
- a visszatért érték összehasonlítható másik értékkel (==, !=), illetve a `name()` metódusával kiíratható a típus neve

```
// typeid
#include <iostream>
#include <typeinfo>
using namespace std;

int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different types:\n";
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

typeid

- Osztályok esetén az operátor itt is az RTTI-t használja a dinamikus objektumok típusának meghatározására,
- Polimorf osztály típusú kifejezés esetén az eredmény a hierarchiában legalul (most derived) lévő teljes objektum:

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class Base { virtual void f(){} };
class Derived : public Base {};

int main () {
  try {
    Base* a = new Base;
    Base* b = new Derived;
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
    cout << "*a is: " << typeid(*a).name() << '\n';
    cout << "*b is: " << typeid(*b).name() << '\n';
  } catch (exception& e) { cout << "Exception: " << e.what() << '\n'; }
  return 0;
}
```

typeid

- a `name()` függvény által visszaadott szöveg a fordító implementációjától függ,
- a példában láthatjuk, hogy mutatók esetén a mutató típusát adja vissza,
- objektumok esetén pedig az objektum dinamikus típusát kapjuk,
- ha egy mutató a paraméter, amit a * dereferencia operátor előz meg és a mutató értéke `null`, akkor az operátor egy `bad_typeid` kivételt dob.