Objektumorientált programozás

Objektumalapú programozás a C++ programozási nyelvben

Az osztály fogalma

Darvay Zsolt

Osztályok, kivételkezelés és sablonok

11. Az osztály fogalma
12. Kivételkezelés
13. Sablonok

11. Az osztály fogalma

Áttekintés

11.1. Adatvédelem moduláris programozással
11.2. Absztrakt adattípus
11.3. Osztályok deklarálása
11.4. Az objektumok tagjaira való hivatkozás és a this mutató
11.5. A konstruktor
11.6. A destruktor

11.1. Adatvédelem moduláris programozással

- A függvényeken kívül deklarált statikus változókat használjuk.
- Egy állományba (modulba) kerülnek az adatok, mint statikus változók, és a rájuk vonatkozó függvények.
- A statikus változók védettek. Más modulból nem érhetők el akkor sem, ha az illető állományban egy extern típusú deklaráció szerepel. Ez a fajta védettség a C programozási nyelvre jellemző.

Példa vektorokra vonatkozó modulra

- Két állomány:
  - vektor1.cpp
  - fo1.cpp
- A két állomány ugyanabba a projektbe kell kerüljön. A projekt akkor is lefordítható, ha a vektor1.cpp állomány helyett a neki megfelelő lefordított kód kerül a projektbe (vektor1.obj).

A vektor1.cpp állomány

```
#include <iostream>
using namespace std;
static int* elem;
static int dim;
// ...
```

Az init függvény

```
void init(int *e, int d)
{
   elem = new int[d];
   dim = d;
   for(int i=0; i < dim; i++)
        elem[i] = e[i];
}
```

A felszabadit és negyzetreemel függvények

```
void felszabadit() {
   delete []elem;
}
void negyzetreemel() {
   for(int i = 0; i < dim; i++)
        elem[i] *= elem[i];
}
```

A kiir függvény

```
void kiir() {
   for(int i = 0; i < dim; i++)
        cout << elem[i] << '\t';
   cout << endl;
}
```

A fo1.cpp állomány

```
void init( int*, int);          // esetleg extern
void felszabadit();
void negyzetreemel();
void kiir();
//extern int* elem;
void main() {  // ...
}
```

A fő függvény

```
void main() {
  int x[5] = {1, 2, 3, 4, 5};
  init(x, 5);
  negyzetreemel();
  kiir();                          // 1  4  9  16  25
  felszabadit();
}
```

11.2. Absztrakt adattípus

- Olyan struktúra (adatszerkezet), amely az adatokon kívül a rájuk vonatkozó műveleteket is tartalmazza.

```
struct név {
        // adatok
        // műveletek
};
```

adattagok

tagfüggvények

A tagfüggvények definíciója

- Általában a tagfüggvények deklarációját helyezzük el a struktúrán belül. A definíció esetén a névterekhez hasonlóan a hatókör operátort használjuk.
- Ha egy tagfüggvény definícióját a struktúrán belül helyezzük el, akkor az inline függvény lesz (az inline kulcsszót nem kell megadni).

Példa vektorokra vonatkozó absztrakt adattípusra

```
struct vektor {
   int *elem;
   int dim;
   void init(int *e, int d);
   void felszabadit();
   void negyzetreemel();
   void kiir();
};
```

adattagok

tagfüggvények

Az init függvény

```
void vektor::init(int *e, int d)
{
   elem = new int[d];
   dim = d;
   for(int i=0; i < dim; i++)
        elem[i] = e[i];
}
```

A felszabadit és negyzetreemel függvények

```
void vektor::felszabadit() {
   delete []elem;
}
void vektor::negyzetreemel() {
   for(int i = 0; i < dim; i++)
        elem[i] *= elem[i];
}
```

A kiir függvény

```
void vektor::kiir()  {
   for(int i = 0; i < dim; i++)
         cout << elem[i] << '\t';
   cout << endl;
}
```

- A fő függvényben a vektor adatszerkezetnek egyszerre több példányával dolgozhatunk:

```
vektor v1, v2;
```

A fő függvény

```
void main() {
   int x[]={1, 2, 3, 4, 5};
   vektor v1;
   v1.init(x, 5);
   v1.negyzetreemel();
   v1.kiir();                  // 1  4  9  16  25
   v1.elem[1] = 100;           // nincs védelem
   v1.kiir();                  // 1  100  9  16  25
   v1.felszabadit();
}
```

Az inicializálás megvalósítása konstruktorral

- Az absztrakt adattípusok esetén az adattagok inicializálását általában sajátos függvényekkel, úgynevezett konstruktorokkal végezzük.
- A konstruktor neve meg kell egyezzen az absztrakt adattípus nevével.
- Az absztrakt adattípus egy példányának létrehozásakor mindig végre lesz hajtva egy konstruktor.

Absztrakt adattípusok destruktora

- Az absztrakt adattípus példányainak megszűnésekor mindig végre lesz hajtva egy sajátos tagfüggvény, amit destruktornak nevezünk.
- A destruktor neve a ~ karakterrel kezdődik, ez után pedig az absztrakt adattípus neve következik.
- Sem a konstruktor, sem a destruktor nem térít vissza értéket (a void típust sem szabad megadni).

Példa absztrakt adattípusra, konstruktorral és destruktorral

```
struct vektor {
   int *elem;
   int dim;
   vektor(int *e, int d);
   ~vektor() { delete [] elem; }    // inline
   void negyzetreemel();
   void kiir();
};
```

A konstruktor

```
vektor::vektor(int *e, int d)
{
   elem = new int[d];
   dim = d;
   for(int i=0; i < dim; i++)
        elem[i] = e[i];
}
```

A negyzetreemel tagfüggvény és a destruktor

```
void vektor::negyzetreemel()
{
   for(int i = 0; i < dim; i++)
        elem[i] *= elem[i];
}
```

- A destruktort a struktúrán belül definiáltuk, ezért ez inline függvény lesz.

A kiir függvény

```
void vektor::kiir()
{
   for(int i = 0; i < dim; i++)
        cout << elem[i] << '\t';
   cout << endl;
}
```

A fő függvény

```
void main() {
   int x[]={1, 2, 3, 4, 5};
   vektor v1(x, 5);            // konstruktormeghívás
   v1.kiir();
   v1.negyzetreemel();
   v1.kiir();
   v1.elem[1] = 100;           // nincs védelem
   v1.kiir();
}              // a destruktort automatikusan meghívja
```

Absztrakt adattípusok és a moduláris programozás

- Az absztrakt adattípust egy olyan struktúra segítségével értelmezzük, amely az adatokon kívül a rájuk vonatkozó műveleteket is tartalmazza.
- Ez a változat a moduláris programozásnál jobb, abból a szempontból, hogy egyszerre több példánnyal tudunk dolgozni.
- Meg kell valósítni viszont az adatok védelmét, amire a moduláris programozás esetén lehetőség volt.

A hozzáférés szabályozása

- Hozzáférés módosítókkal történik. Ezek a következők:
  - privát (private)
  - védett (protected)
  - nyilvános (public)
- A hozzáférés módosítókat, mint címkéket adjuk meg (egy kettőspont következik utánuk).

Példa hozzáférés módosítókra

```
struct név {
private:
        // privát tagok
public:
        // nyilvános tagok
protected:
        // védett tagok
};
```

A tagok elérése

- Egy hozzáférés módosító az összes utána következő tagra vonatkozik, egészen a következő hozzáférés módosítóig, ha van olyan.
- A struktúra esetén a tagok alapértelmezés szerint nyilvánosak.
- Ha vannak olyan tagok, amelyek nem nyilvánosak, akkor általában osztályokat használunk.

11.3. Osztályok deklarálása

- Az osztály egy olyan absztrakt adattípus, amely lehetőséget teremt a tagok elérésének szabályozására, és alapértelmezés szerint minden tag privát.
- A struktúra olyan osztálynak tekinthető, amelyben a tagok alapértelmezett elérhetősége nyilvános.
- Az osztályt ugyanúgy deklaráljuk, mint a struktúrát, csak a struct kulcsszó helyett class lesz.

Példa osztályra, hozzáférés módosítókkal

```
class név {
        // privát tagok (alapértelmezés)
public:
        // nyilvános tagok
protected:
        // védett tagok
};
```

A privát tagok elérése

- A privát tagokat az illető osztály tagfüggvényeiben el lehet érni.
- Ezen kívül az osztály barát (friend) függvényeiben is elérhetők a privát tagok.
- A privát tagok máshol nem elérhetők.

A nyilvános tagok elérése

- A nyilvános tagok bárhol elérhetők.
- Mivel a struktúra tagjai alapértelmezés szerint nyilvánosak, ezek bárhol elérhetők, ha a struktúrán belül nem használtunk hozzáférés módosítókat.

A védett tagok elérése

- A védett tagok elérhetők:
  - az osztály tagfüggvényeiben;
  - az osztály barát függvényeiben;
  - az illető osztályból származtatott osztály tagfüggvényeiben;
  - az illető osztályból származtatott osztály barát függvényeiben.

Példa osztályra

```
class vektor {
   int *elem;
   int dim;
public:
   vektor(int *e, int d);
   ~vektor() { delete [] elem; }
   void negyzetreemel();
   void kiir();
};
```

privát

nyilvános

A konstruktor

```
vektor::vektor(int *e, int d)
{
   elem = new int[d];
   dim = d;
   for(int i=0; i < dim; i++)
        elem[i] = e[i];
}
```

A negyzetreemel tagfüggvény és a destruktor

```
void vektor::negyzetreemel()
{
   for(int i = 0; i < dim; i++)
        elem[i] *= elem[i];
}
```

- A destruktort a struktúrán belül definiáltuk, ezért ez inline függvény lesz.

A kiir függvény

```
void vektor::kiir()
{
   for(int i = 0; i < dim; i++)
        cout << elem[i] << '\t';
   cout << endl;
}
```

A fő függvény

```
void main() {
   int x[]={1, 2, 3, 4, 5};
   vektor v1(x, 5);   // konstruktor: nyilvános
   v1.negyzetreemel();     // nyilvános
   v1.kiir();              // nyilvános
   // v1.elem[1] = 100;    // hiba: privát
}                          // a destruktor: nyilvános
```

Az objektum fogalma

- Egy osztály példányait objektumoknak nevezzük. Például:

  vektor v1(x, 5);

- A v1 egy objektum.
- Az osztály egy típus.
- Az objektum általában egy változó, amelynek a típusa az illető osztály.

Barát függvények

- Egy osztály barát függvénye nem tagfüggvénye az osztálynak.
- A barát függvény deklarációját az osztálydeklaráción belül kell elhelyezni.
- A függvényt a szokásos módon deklaráljuk, de a deklaráció a friend minősítővel kezdődik.
- A barát függvény definíciója nem kell tartalmazza a friend minősítőt.
- Ha lehet a barát függvények használatát kerülni kell.

Barát osztályok

- Ha egy osztály összes tagfüggvénye barát függvénye egy másik osztálynak, akkor a teljes osztályt mint barát osztályt jelenthetjük be.
- A „barát” reláció nem szimmetrikus, és nem tranzitív.
- Csak egymással szorosan összekapcsolódó osztályokat érdemes barátoknak deklarálni.

Példa barát osztályra

```
class oszt1;          // előzetes, nem teljes deklaráció
class oszt2 {
   // ...
   friend class oszt1;      // oszt1 barát osztálya
   // ...                   // oszt2-nek.
};
```

- Az oszt1 osztály összes tagfüggvénye barát függvénye lesz az oszt2 osztálynak.

11.4. Az objektumok tagjaira való hivatkozás és a this mutató

- A tagokra való hivatkozás a struktúrák esetén használatos tagkiválasztó operátorral (.), illetve a struktúra-mutató operátorral (->) történik, függetlenül attól, hogy adattagról, vagy tagfüggvényről van szó.

Példa a tagokra való hivatkozásra

```
vektor v1(x, 5);
vektor *p = &v1;
v1.kiir();
p->kiir();
```

- A tagfüggvények belsejében direkt módon hivatkozhatunk az osztály tagjaira, nincs szükség tagkiválasztó, vagy struktúra-mutató, operátorra.

A this mutató

- A tagfüggvények belsejében a tagokra való hivatkozás a this mutató segítségével történik.
- Minden tag helyett this->tag lesz.
- A this mutató az aktuális objektumra mutat.
- A mi esetünkben a this a v1 címével, illetve a p-vel egyezik meg.
- A this mutatót explicit módon is használhatjuk, ha erre szükség van.

Példa a this mutatóra

```
void vektor::negyzetreemel() {
   for(int i = 0; i < dim; i++)
        elem[i] *= elem[i];
}
```

- Ezt a rendszer a következővel helyettesíti:

```
void vektor::negyzetreemel() {
   for(int i = 0; i < this->dim; i++)
        this->elem[i] *= this->elem[i];
}
```

11.5. A konstruktor

11.5.1. Objektum inicializálása konstruktorral

11.5.2. A konstruktor meghívása, ha az adattag egy objektum

11.5.1. Objektum inicializálása konstruktorral

- Az objektum létrehozását konstruktorral végezzük.
- A konstruktor neve meg kell egyezzen az osztály nevével.
- Mivel a függvények túlterhelhetők, egy osztálynak több konstruktora is lehet, ha a paraméterlisták különböznek.
- A konstruktor nem térít vissza értéket (a void típust sem szabad megadni).

Példa több konstruktorra

```
class szemely {
  char* cs_nev;
  char* sz_nev;
public:
   szemely();                    //alapértelmezett konstruktor
   szemely(char* cs_n, char* sz_n);
   szemely(const szemely& sz);   // másoló konstruktor
   ~szemely();
   void kiir();
};
```

Alapértelmezett konstruktor

- Más néven: alapértelmezés szerinti konstruktor.
- Ha a konstruktor formális paramétereinek listája üres, akkor alapértelmezett konstruktorról beszélünk.
- Ha egy osztálynak van alapértelmezett konstruktora, akkor létrehozható olyan objektum, amely nem tartalmaz inicializáló aktuális paraméterekből álló listát.
- Ez akkor is lehetséges, ha olyan konstruktorunk van, amelynek az összes formális paramétere kezdeti értékkel van ellátva.

Ha nem adunk meg konstruktort ...

- Ha nincs, a programozó által definiált konstruktor, akkor a rendszer létrehoz egy alapértelmezett konstruktort, és ezt hívja meg az objektumok létrehozásakor.
- Ha van, a programozó által definiált konstruktor, de az nem alapértelmezett, a rendszer semmiképpen nem hoz automatikusan létre alapértelmezett konstruktor (ezt is definiálni kell).

Másoló konstruktor

- Célja egy objektum inicializálása egy ugyanolyan típusú objektum segítségével.
- Általában az

osztálynév(const osztálynév & objektum);

alakban deklaráljuk. A const arra utal, hogy a paraméterként megadott objektum nem változik.

Az alapértelmezett konstruktor

```
szemely::szemely() {
  cs_nev = new char[1];
  *cs_nev = 0;                    // 0 és ’\0’ ugyanaz
  sz_nev = new char[1];
  *sz_nev = 0;
  cout << "Alapertelmezett konstruktor\n”;
}
```

- A függvény mindkét adattagot az üres karakterlánccal inicializálja.

A hagyományos konstruktor

```
szemely::szemely(char* cs_n, char* sz_n)
{
  cs_nev = new char[strlen(cs_n)+1];
  sz_nev = new char[strlen(sz_n)+1];
  strcpy(cs_nev, cs_n);
  strcpy(sz_nev, sz_n);
  cout << "Hagyomanyos konstruktor\n”;
}
```

A másoló konstruktor

```
szemely::szemely(const szemely& x)
{
  cs_nev = new char[strlen(x.cs_nev)+1];
  strcpy(cs_nev, x.cs_nev);
  sz_nev = new char[strlen(x.sz_nev)+1];
  strcpy(sz_nev, x.sz_nev);
  cout << "Masolo konstruktor\n";
}
```

A destruktor és a kiir tagfüggvény

```
szemely::~szemely() {
  cout << "Destruktor\n";
  delete[] cs_nev;
  delete[] sz_nev;
}
void szemely::kiir() {
  cout << sz_nev << ' ' << cs_nev << endl;
}
```

A fő függvény

```
void main() {
  szemely A;              //alapértelmezett konstruktor
  A.kiir();
  szemely B("Stroustrup", "Bjarne");
  B.kiir();
  szemely *C = new szemely("Kernighan","Brian");
  C->kiir();
  delete C;
}
```

Kimenet

Alapertelmezett konstruktor

Hagyomanyos konstruktor
Bjarne Stroustrup
Hagyomanyos konstruktor
Brian Kernighan
Destruktor
Destruktor
Destruktor

Ha a másoló konstruktor hiányzik ...

- Ha a programozó nem definiál másoló konstruktort, akkor a rendszer létrehoz egy másoló konstruktort, amely az adattagok bitenkénti másolását végzi.
- A bitenkénti másolás általában akkor ad helyes eredményt, ha az osztálynak nincsen mutató típusú adattagja.

Egy másik fő függvény

```
void main() {
  szemely B("Stroustrup", "Bjarne");
  B.kiir();
  szemely D(B);  // ugyanaz: szemely D = B;
                 // másoló konstruktor
  D.kiir();
}
```

- Ha nem definiálunk másoló konstruktort, akkor kétszer próbálja meg felszabadítani ugyanazt a memóriaterületet, ezert futási hibával megáll.

Kimenet

Hagyomanyos konstruktor
Bjarne Stroustrup
Masolo konstruktor
Bjarne Stroustrup
Destruktor
Destruktor

Magyar nevek kiírása

- A szemely osztályt kiegészítjük egy olyan tagfüggvénnyel, amely a családnevet írja ki először.

```
class szemely {
        // …
public:
        void kiir_hun();
        // …
};
```

A kiir_hun függvény

```
void szemely::kiir_hun()
{
  cout << cs_nev << ' '
       << sz_nev << endl;
}
```

A másoló konstruktor meghívása

- A rendszer akkor hívja meg, ha:
  - ugyanolyan típusú objektummal inicializálunk;
  - egy függvénynek objektum típusú paramétere van;
  - egy függvény objektum típust térít vissza.
- Ezért, ha van pointer típusú adattag, akkor a másoló konstruktort definiálnunk kell akkor is, ha nincs szándékunkban ugyanolyan típusú objektummal inicializálni.

A vektor osztály kiegészítése az osszead tagfüggvénnyel

- Az osztálydeklaráció
- A másoló konstruktor
- Az osszead tagfüggvény
- A fő függvény

Az osztálydeklaráció

```
class vektor {    int *elem;
                  int dim;
public:
        vektor(int *e, int d);
        vektor(const vektor &v); //masoló konstruktor
        ~vektor() { delete [] elem; }
        void negyzetreemel();
        vektor osszead(vektor& v); // összeadás
        void kiir();
};
```

A másoló konstruktor

```
vektor::vektor(const vektor &v)
{
   dim = v.dim;
   elem = new int[dim];
   for(int i=0; i < dim; i++)
        elem[i] = v.elem[i];
}
```

Az osszead tagfüggvény

```
vektor vektor::osszead(vektor& v) {
  if (dim != v.dim) {
      cerr << "Hiba: kulonbozo dimenzio"; exit(1); }
  int* x = new int[dim];
  for(int i = 0; i < dim; i++)
    x[i] = elem[i] + v.elem[i];
  vektor t(x, dim);
  delete [] x;
  return t;
}
```

A fő függvény

```
void main() {
   int x[]={1, 2, 3, 4, 5};
   vektor v1(x, 5);
   int y[]={2, 4, 6, 8, 10};
   vektor v2(y, 5);
   v1.osszead(v2).kiir();  // 3 6 9 12 15
}
```

11.5.2. A konstruktor meghívása, ha az adattag egy objektum

- Egy osztály tartalmazhatja más osztályok objektumait adattagként. Például:

```
class oszt {
        oszt_1 ob_1;
        oszt_2 ob_2;
        …
        oszt_n ob_n;
};
```

A konstruktor fejléce

- Ebben az esetben az „oszt” osztály konstruktorának a fejléce a következő alakú lesz:

  oszt(argumentumlista) : objektumlista

- Az objektumlista a következő alakú:

  ob_1(arglista_1), ob_2(arglista_2), ..., ob_n(arglista_n)

- Az argumentumlista az „oszt” osztály konstruktorában a formális paraméterek listája.
- Az arglista_i az ob_i osztály konstruktorában a formális paraméterek listája.

Az objektumlista

- Hiányoznak belőle azok az objektumok, amelyek nem rendelkeznek a programozó által definiált konstruktorral.
- Ugyancsak hiányozhatnak azok az objektumok, amelyekre az alapértelmezett konstruktort, vagy egy olyan konstruktort szeretnénk meghívni, amelynek az összes paramétere inicializálva van.

A konstruktorok meghívása

- Ha egy osztálynak egyik adattagja egy objektum, akkor először ennek az objektumnak a konstruktorát hívja meg a rendszer, majd ezt követően lesz végrehajtva az osztály konstruktorának a törzse.

Példa olyan adattagra, amely objektum

```
class hazaspar {
  szemely ferj;
  szemely feleseg;
public:
  hazaspar()        // alapértelmezett konstruktor,
  {                 // a ferj, es feleseg objektumok
  }                 // alapértelmezés szerinti
  // ...            // konstruktorát hívja meg
};
```

A többi tagfüggvény

```
 // ...
 hazaspar(szemely& aferj, szemely& afeleseg);
 hazaspar(char* cs_ferj, char* sz_ferj,
                char* cs_feleseg, char* sz_feleseg):
                ferj(cs_ferj, sz_ferj),
                feleseg(cs_feleseg, sz_feleseg)
                { }        // definíció
  void kiir();
  void kiir_hun();
};
```

Definíció az osztályon kívül

```
inline hazaspar::hazaspar(szemely& aferj,
        szemely& afeleseg):
        ferj(aferj), feleseg(afeleseg)
{
}
```

A kiir és kiir_hun függvények

```
void hazaspar::kiir() {
  cout << "ferj: ";        ferj.kiir();
  cout << "feleseg: ";     feleseg.kiir();
}
void hazaspar::kiir_hun() {
  cout << "ferj: ";        ferj.kiir_hun();
  cout << "feleseg: ";     feleseg.kiir_hun();
}
```

A fő függvény

```
void main() {
  szemely Ady("Ady","Endre");
  szemely Csinszka("Boncza","Berta");
  hazaspar h(Ady, Csinszka);
  h.kiir_hun();
  hazaspar h2("Petofi", "Sandor", "Szendrei", "Julia");
  h2.kiir_hun();
  hazaspar XY;
  XY.kiir();
}
```

A kimenet

ferj: Ady Endre

feleseg: Boncza Berta

ferj: Petofi Sandor

feleseg: Szendrei Julia

ferj:

feleseg:

11.6. A destruktor

- Ha egy objektum megszűnik, a rendszer automatikusan végrehajtja a destruktort.
- A destruktor neve a ~ karakterrel kezdődik, és ez után az osztály neve következik.
- A destruktor nem térít vissza értéket (a void típust sem szabad megadni).

A destruktor meghívása

- Egy globális objektum destruktora a main függvény végén az exit függvény részeként lesz végrehajtva.
- Ezért nem szabad az exit függvényt meghívni a destruktorban, mivel ez végtelen ciklust eredményezhet.
- Egy helyi objektum destruktorát akkor hívja meg a rendszer, ha annak a blokknak a végére értünk, amelyben definiálva volt.
- A dinamikus módon létrehozott objektum destruktorát a delete operátoron keresztül hívja meg a rendszer.

Példa destruktorra

```
#include <stdio.h>
#include <string.h>
class kiiras {      // kiirja, hogy mit hívott meg
      char* nev;
public:
      kiiras(char* n);
      ~kiiras();
};
```

A konstruktor

```
kiiras::kiiras(char* n)
{
  nev = new char[strlen(n)+1];
  strcpy(nev, n);
  printf("Letrehoztam: %s\n", nev);
}
```

A destruktor

```
kiiras::~kiiras()
{
  printf("Felszabaditottam: %s\n", nev);
  delete nev;
}
```

Helyi és globális objektumok

```
void fuggv()
{
    printf("Fuggvenymeghivas.\n");
    kiiras helyi("HELYI");
}
kiiras globalis("GLOBALIS");
```

Dinamikus objektum

```
void main() {
  kiiras* dinamikus = new kiiras("DINAMIKUS");
  fuggv();
  printf("Folytatodik a fo fuggveny.\n");
  delete dinamikus;
}
```

A kimenet

Letrehoztam: GLOBALIS
Letrehoztam: DINAMIKUS
Fuggvenymeghivas.
Letrehoztam: HELYI
Felszabaditottam: HELYI
Folytatodik a fo fuggveny.
Felszabaditottam: DINAMIKUS
Felszabaditottam: GLOBALIS