Objektumorientált programozás

Objektumalapú programozás a C++ programozási nyelvben

Operátorok túlterhelése

Darvay Zsolt

Túlterhelés, konverzió és alosztályok

14. Operátorok túlterhelése
15. A programozó által definiált típuskonverzió
16. Az objektumorientált programozási módszer

14. Operátorok túlterhelése

Áttekintés

14.1. A túlterhelés általános módszere
14.2. Értékadó operátorok túlterhelése
14.3. Léptető operátorok túlterhelése
14.4. A << operátor túlterhelése
14.5. A new és delete operátorok túlterhelése
14.6. A tömbindexelés operátor túlterhelése
14.7. A függvénymeghívás operátor túlterhelése
14.8. A struktúra-mutató operátor túlterhelése

14.1. A túlterhelés általános módszere

- Az eddigiekben definiált vektor osztály esetén a vektorok összeadását egy tagfüggvény segítségével végeztük. Ha v1 és v2 két vektor, akkor az összegüket így írhatjuk ki:

  v1.osszead(v2).kiir();

- E helyett jobb lenne, ha a következőt írhatnánk:

  cout << v1 + v2;

- Ezt a << és a + operátorok túlterhelésével valósíthatjuk meg.

Általános szabályok

- Nem lehet új operátort definiálni, csak a meglévő operátorkészletet lehet túlterhelni (új jelentéssel ellátni).
- Vannak operátorok, amelyek nem terhelhetők túl, például a hatókör (::), a tagkiválsztó (.) és a feltételes (?:) operátor.
- A túlterhelés által nem változtatható meg az operátor jellege, tehát az hogy bináris vagy unáris, illetve a kiértékelés iránya sem.

A túlterhelés megvalósítása

- Sajátos tagfüggvénnyel, vagy barát függvénnyel végezzük.
- A függvény neve az operator kulcsszóval kezdődik, és ezt a túlterhelendő operátor követi. Ezt a kettőt egy fehér karakter választhatja el.

A formális paraméterek száma

- Egyértelműen meghatározható, attól függően, hogy az illető operátor unáris vagy bináris, illetve a túlterhelést megvalósító függvény tagfüggvény vagy barát függvény.
- Ezt a következő táblázat szemlélteti:

	tag	barát
unáris	0	1
bináris	1	2

A + operátor túlterhelése a vektor osztály esetén

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

A + operátor

```
vektor vektor::operator +(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

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

14.2. Értékadó operátorok túlterhelése

- Ha a programozó nem terheli túl az értékadás operátort (=), akkor alapértelmezett értékadó operátort hoz létre a rendszer, amely a másoló konstruktorhoz hasonlóan, egy bitenkénti átmásolást végez.
- Ez az alapértelmezett értékadó operátor általában akkor működik helyesen, ha az osztály nem rendelkezik mutató típusú adattaggal.
- Ellenkező esetben az értékadó operátort túl kell terhelni.

Olyan osztály, amelyre nem kell túlterhelni az értékadó operátort

```
#include <iostream>
using namespace std;
class tort {
    int sz; // számláló
    int n;  // nevező
public:
    tort(int sz1 = 0, int n1 = 1);
    tort operator *(tort& r);   //nem egyszerüsít
    void kiir();
};
```

A konstruktor

```
inline tort::tort(int sz1, int n1)
{
  sz = sz1;
  n = n1;
}
```

A * operátor

```
inline tort tort::operator *(tort& r)
{
   return tort(sz*r.sz, n*r.n);
}
```

A kiir tagfüggvény

```
inline void tort::kiir()
{
   if (n)
        cout << sz << " / " << n << endl;
   else
        cerr << "hibas tort";
}
```

A fő függvény

```
int main() {
   tort x(3, 5);
   tort y(2, 7);
   tort z;
   z = x * y;
   z.kiir();   //    6 / 35
}
```

Olyan osztály, amelyre túl kell terhelni az értékadó operátort

- Az általunk eddig leírt „vektor” osztálynak van egy mutató típusú adattagja (elem), ezért túl kell terhelni az értékadó operátort.
- Ha ezt nem tesszük meg, akkor két v1 és v2 vektor esetén a

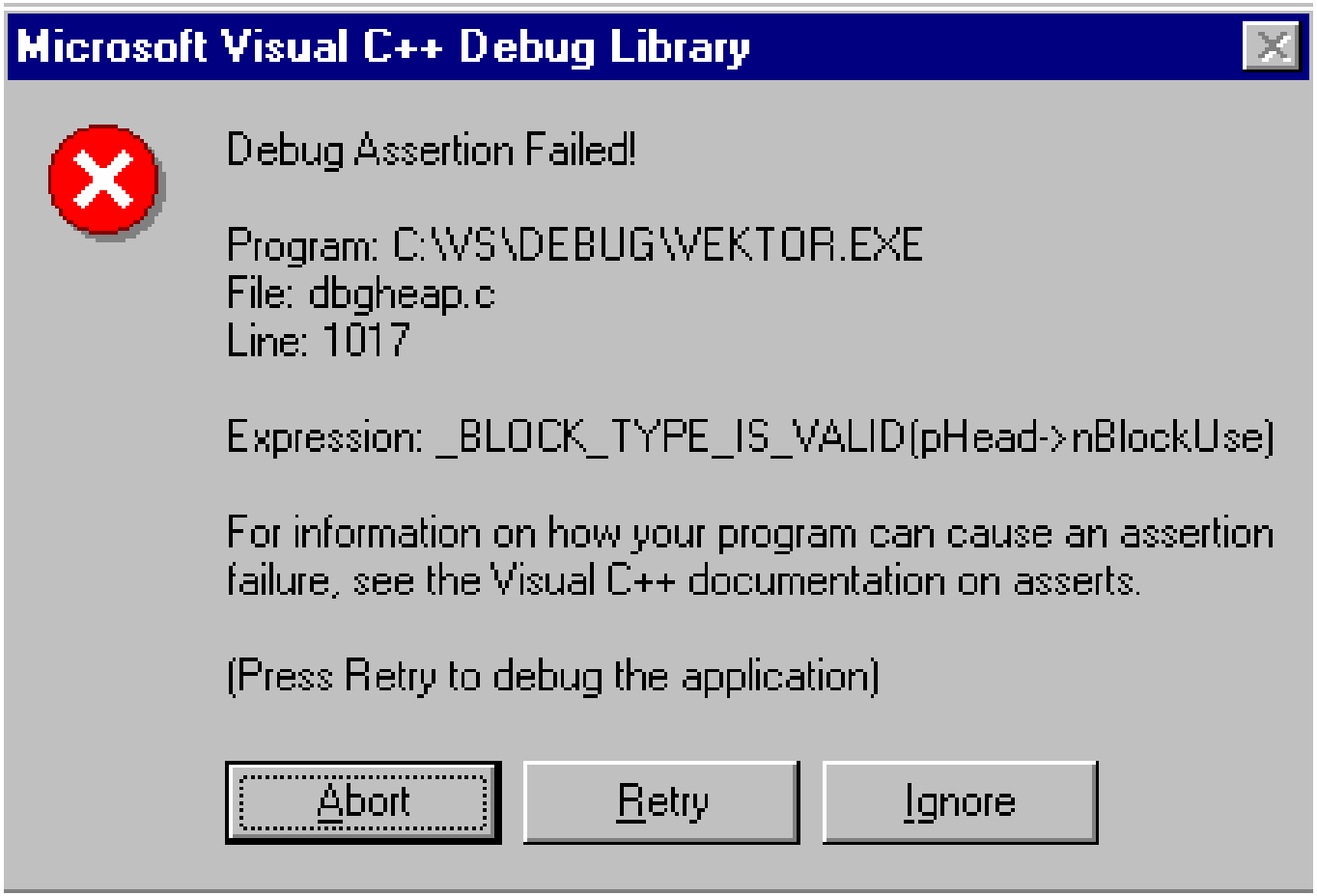
  v2 = v1;

- értékadás hibához vezet. Ez a hiba futtatás közben fog jelentkezni, amikor az objektumok destruktorát meghívja a rendszer.

A fő függvény

```
int main() {
   int x[] = { 1, 2, 3, 4, 5 };
   vektor v1(x, 5);
   int y[] = { 2, 4, 6, 8, 10 };
   vektor v2(y, 5);
   v2 = v1;
}
```

A hiba Visual C++ fordító esetén



A hiba oka

- A bitenkénti másolás miatt a két vektor elem adattagja meg fog egyezni, tehát azonos címre fog hivatkozni.
- A fő függvény végén a rendszer automatikusan végrehajtja a v1 és v2 objektumok destruktorát, ezért az a memóriaterület, amelyre az elem adattag mutat, kétszer lesz felszabadítva. Ez okozza a hibát.
- Egyes C++ fordítók nem fognak hibát jelezni.

A vektor osztály

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

Az = operátor

```
vektor& vektor::operator =(const vektor& v) {
   if (this != &v) {
        delete[] elem;
        dim = v.dim;
        elem = new int[dim];
        for (int i = 0; i < dim; i++)
                elem[i] = v.elem[i];
   }
return *this;
}
```

A fő függvény

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

A többi értékadó operátor túlterhelése

- Az op= értékadó operátorokat, ahol op egy tetszőleges bináris aritmetikai, vagy bitenkénti operátor nem terheli túl alapértelmezés szerint a rendszer.
- Ezért ezeket, ha használni akarjuk, mindenképpen túl kell terhelni.
- Ha az op operátor már túl van terhelve, akkor ezt hívhatjuk meg, és az aktuális objektumra a *this segítségével hivatkozhatunk.

A vektor osztály

```
class vektor {
    int *elem;
    int dim;
public:
    vektor(int *e, int d);
    vektor(const vektor &v);
    ~vektor() { delete[] elem; }
    vektor operator +(const vektor& v);
    vektor& operator =(const vektor& v);
    vektor& operator +=(const vektor& v);
    void kiir();
};
```

A += operátor

```
vektor& vektor::operator +=(const vektor& v)
{
   return *this = *this + v;
}
```

- Mivel a v formális paraméter const minősítővel van deklarálva, a + operátor esetén is jelen kell legyen a const minősítő.

Ha a const minősítő hiányzik

- Ha a + operátor esetén a formális paraméter a const minősítő nélkül van deklarálva, akkor a += operátor így módosulhat:

```
vektor& vektor::operator +=(const vektor& v)
{
   return *this = *this + const_cast<vektor&>(v);
}
```

- A const_cast esetén a cél típus mutató, vagy referencia kell legyen.

A fő függvény

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

A másoló konstruktor és az értékadó operátor

```
vektor v2(v1);        // másoló konstruktor
vektor v2 = v1;       // másoló konstruktor

vektor v2;    // alapértelmezett konstruktor
v2 = v1;      // értékadó operátor
```

14.3. Léptető operátorok túlterhelése

- A léptető operátort előtagként és utótagként is használhatjuk.
- Ha az operátort tagfüggvénnyel terheljük túl, akkor a formális paraméterek listája üres kell legyen.
- Az üres paraméterlistával rendelkező tagfüggvény az előtagként használt léptető operátornak felel meg, de az operátort utótagként is használhatjuk. Ebben az esetben figyelmeztető üzenet („warning”) jelenik meg.

Az előtag és utótag megkülönböztetése

- Ha azt szeretnénk, hogy az előtagként, illetve utótagként, használt léptető operátorok különböző függvényeknek feleljenek meg, akkor egy másik tagfüggvényt is kell definiáljunk, amely egyetlen int típusú paraméterrel rendelkezik.
- Ez a tagfüggvény az utótagként használt léptető operátor esetén lesz végrehajtva.

Az int típusú paraméter

- Az int típusú paramétert nem használjuk semmire, csak az a szerepe, hogy megkülönböztesse a két függvényt.
- A formális paraméter nevét sem kell megadni ebben az esetben.

Összehasonlítás a standard típusokkal

- Standard típusok esetén az előtagként megadott léptető operátort előbb végrehajtja a rendszer majd a kapott értéket használja a kifejezésben. Utótag esetén a változó értékét használjuk a kifejezésben, majd módosítjuk az értékét.
- Osztályok esetén ez a szabály megszűnik, de azt is megtehetjük, hogy a léptető operátorokat úgy terheljük túl, hogy a standard típusokhoz hasonlóan lehessen őket használni.

Példa léptető operátorra

```
class tort {
   int sz;
   int n;
public:
   tort(int sz1 = 0, int n1 = 1);
   tort& operator++();          // előtag
   tort operator++(int);        // utótag
   void kiir(const char *s);
};
```

A konstruktor

```
inline tort::tort(int sz1, int n1)
{
  sz = sz1;
  n = n1;
}
```

A ++ operátor előtagként

```
tort& tort::operator++()
{
   sz += n;
   return *this;
}
```

- Referenciát térít vissza az aktuális objektumhoz.

A ++ operátor utótagként

```
tort tort::operator++(int) {
   tort t(sz, n);
   sz += n;
   return t;
}
```

- Az aktuális objektumot módosítja, de az előző értékét téríti vissza.

A kiir tagfüggvény

```
void tort::kiir(const char *s)
{
   cout << s << sz << " / " << n << endl;
}
```

- Az s karakterlánc-literált írja ki, majd ezt követően a törtet. Végül új sorra tér.

A fő függvény

```
int main()
{
   tort t1(3, 4);
   t1.kiir("  t1 = ");
   tort t2(3, 4);
   t2.kiir("  t2 = ");
   tort t3;
   t3.kiir("  t3 = ");
   //  ...
```

A ++ operátor meghívása

```
    // ...
    cout << "A t3 = t1++; vegrehajtasa utan:\n";
    t3 = t1++;
    t3.kiir("  t3 = ");
    t1.kiir("  t1 = ");
    cout << "A t3 = ++t2; vegrehajtasa utan:\n";
    t3 = ++t2;
    t3.kiir("  t3 = ");
    t2.kiir("  t2 = ");
}
```

A kimenet

t1 = 3 / 4
t2 = 3 / 4
t3 = 0 / 1

A t3 = t1++; vegrehajtasa utan:

t3 = 3 / 4
t1 = 7 / 4

A t3 = ++t2; vegrehajtasa utan:

t3 = 7 / 4
t2 = 7 / 4

14.4. A << operátor túlterhelése

- Szabványos típusokra a << operátor túl van terhelve.
- A cout az ostream osztály egy objektuma.
- Saját típusokra ezt mi tehetjük meg az

ostream& operator <<(ostream& s, const oszt &o);

- alakú függvénnyel, ahol oszt az illető típust definiáló osztály neve.
- Ha ez az operátor az oszt védett tagjaira kell hivatkozzon, akkor ezt általában az oszt egy nyilvános tagfüggvényén keresztül valósíthatjuk meg.

Példa a << operátor túlterhelésére

```
#include <iostream>
using namespace std;
class tort {
   int sz;
   int n;
public:
   tort(int a, int b) { sz = a; n = b; }
   ostream& kiir(ostream& s) const;
};
```

A kiir tagfüggvény

```
ostream& tort::kiir(ostream& s) const
{
   s << sz << " / " << n;
   return s;
}
```

- A const azt jelzi, hogy a kiir egy konstans függvény, amely nem módosíthatja az adattagokat.

A << operátor

```
ostream& operator<<(ostream& s, const tort &t)
{
   return t.kiir(s);
}
```

A fő függvény

```
int main() {
   tort t1(3, 5);
   cout << t1 << endl;
   return 0;
}
```

Kimenet:

3 / 5

14.5. A new és delete operátorok túlterhelése

- A new és delete operátorok egy szabványos (globális) túlterheléssel rendelkeznek.
- Ha osztályokra alkalmazzuk, akkor a new operátor mindig meghív egy konstruktort:
- new oszt // alapértelmezett konstruktor
- new oszt(paraméterlista) // a paraméterlistának
  // megfelelő konstruktor
- new oszt[elemszám] //elemszám darab
  // alapértelmezett konstruktor

A destruktor meghívása

- Ha egy objektum számára lefoglalt memóriaterületet szabadítjuk fel, akkor a delete operátor mindig meghív egy vagy több destruktort.
- delete p; // egyszer hívja meg a destruktort
- delete [] p;// a destruktort annyiszor hívja meg,
  // ahány objektum számára foglaltunk le
  // memóriaterületet a new operátorral.

A new operátor túlterhelése

- Statikus tagfüggvény lesz, de a static kulcsszót nem kell megadni.
- A tagfüggvény deklarációja:

  void * operator new(size_t hossz);
- A size_t típus az stdlib.h állományban a

  typedef unsigned int size_t;
- alakban van megadva. A hossz paraméternek nem kell értéket adjunk a függvény meghívásakor. Értékét automatikusan kiszámítja a rendszer az objektum hossza alapján.

A new operátor használata

- Az általunk túlterhelt new operátorra a szokásos alakban, például a

  new osztálynév

- segítségével hivatkozhatunk.
- Ha mégis a szabványos new operátort szeretnénk meghívni, akkor ezt a hatókör operátorral tehetjük meg. Például:

  ::new osztálynév

A delete operátor túlterhelése

- A new operátorhoz hasonlóan, ez is statikus tagfüggvény lesz, de a static kulcsszót ebben az esetben sem kell megadni.
- A tagfüggvény deklarációja:

```
void operator delete(void *p);
```

- A túlterhelt delete operátor meghívása a szokásos módon történik. Például: `delete p;`
- A szabványos delete operátorra ebben az esetben is a hatókör operátorral hivatkozunk:

```
::delete p;
```

A new operátor és a kivételek

- Ha nincs elegendő memória
  - a C++ régebbi változataiban (a Visual C++ 6.0-ban is) a new operátor zérust térített vissza.
  - a C++ szabványnak megfelelően egy std::bad_alloc kivételt vált ki (Visual C++ .NET 2002-től kezdődően). Ha mégsem akarunk kivételt kiváltani, akkor a new(std::nothrow) alakot használhatjuk.

14.6. A tömbindexelés operátor túlterhelése

- Azt szeretnénk, hogy az

  objektum[kifejezés]

- alakban legyen használható.
- A [] operátor túlterhelése egy olyan nem statikus tagfüggvénnyel történik, amely referenciát térít vissza a kiválasztott elemre.
- A függvény deklarációja:

  típus& operator [](index_típus ind);

A tömbindexelés operátor meghívása

- Az operátort

  objektum[kifejezés]

  alakban használhatjuk. A kifejezés típusa az index_típus kell legyen.
- Ez a következőt jelenti:

  objektum.operator [](kifejezés)

Példa tömbindexelés operátorra

```
#include <iostream>
using namespace std;
class tomb {        // egy tömb osztály
   int *elem;       // a tömb elemei
   int szam;        // az elemek száma
public:
   tomb(int *e, int sz);
   int & operator [](int i);
};
```

A konstruktor

```
tomb::tomb(int *e, int sz)
{
   szam = sz;
   elem = new int[szam];
   for (int i = 0; i < szam; i++)
        elem[i] = e[i];
}
```

A tömbindexelés operátor

```
int & tomb::operator [](int i)
{
   if (i < 0 || i >= szam)
        throw "Hiba: nem megfelelo index";
   return elem[i];
}
```

A fő függvény

```
int main()
{
   int t[] = { 10, 20, 30, 40 };
   tomb x(t, 4);
   cout << "Harmadik elem: ";
   cout << x[2] << endl;
   // x egy objektum, mégis alkalmazható a
   // szögletes zárójel operátor
```

A fő függvény

```
    try {
        cout << "Otodik elem: ";
        cout << x[4] << endl;
    }
    catch (const char *s) {
        cout << s << endl;
    }
}
```

A kimenet

Harmadik elem: 30

Otodik elem: Hiba: nem megfelelo index

- Egy hagyományos tömb nem végzett volna az indexekre vonatkozó ellenőrzést.

14.7. A függvénymeghívás operátor túlterhelése

- A () operátort úgy szeretnénk túlterhelni, hogy az:

  objektum(lista)

  alakban lehessen használni, ahol a lista az aktuális paraméterek listája.
- Ez a kifejezés a következőt jelenti:

  objektum.operator ()(lista)

Példa a függvénymeghívás operátor túlterhelésére

```
class tomb {
    int *elem;
    int szam;
public:
    tomb(int *e, int sz);
    int & operator [](int i);
    int & operator()(int x, int y, int m);
};
```

A függvénymeghívás operátor

```
int & tomb::operator()(int x, int y, int m)
{
   return (*this)[x*m + y];
}
```

- Tételezzük fel, hogy a tömb elemeit, soronként, egy m darab oszlopból álló M mátrixba másoljuk át. A függvény az M[x,y] elemnek megfelelő értéket téríti vissza.

A fő függvény

```
int main() {
    int v[] = { 1,2,3,4,5,6,7,8,9 };
    tomb u(v, 9);
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++)
            cout << u(i, j, 3) << ' ';
        cout << endl;
    }
}
```

Kimenet:

```
1 2 3
4 5 6
7 8 9
```

14.8. A struktúra-mutató operátor túlterhelése

- Az objektum->kifejezés a következőt jelenti:

  (objektum.operator ->())->kifejezés

- Ha az objektum.operator ->() egy mutatót térít vissza, akkor az erre vonatkozó struktúra-mutató operátort használjuk. Ha a visszatérített érték egy objektum, akkor erre is túl kell legyen terhelve a -> operátor.

Példa

```
#include <iostream>
using namespace std;
class szemely {
   char nev[20];
   char lakhely[30];
public:
   szemely(const char *nev, const char *lakhely);
   void kiir();
};
```

A szemely osztály konstruktora

```
szemely::szemely(const char *nev, const char *lakhely)
{
   strcpy_s(this->nev, nev);
   strcpy_s(this->lakhely, lakhely);
}
```

Kiírás

```
void szemely::kiir()
{
   cout << nev << endl;
   cout << lakhely << endl;
}
```

Az sz_eletkor osztály

```
class  szEvszam {
   szemely sz;
   int evszam;        // születési év
public:
   szEvszam(szemely sz1, int e1) : sz(sz1)
   {
        evszam = e1;
   }
   szemely * operator ->() { return &sz; }
};
```

A fő függvény

```
int main()
{
   szemely A("Bolyai", "Kolozsvar");
   szEvszam B(A, 1802);
   B->kiir();          //ugyanaz mint: B.sz.kiir();
  // azonban B.sz.kiir(); esetén hibát kapunk
  // az sz privát jellege miatt
}
```