*Az alábbi feladatokat a AUT.BME.HU tanszéki weboldalon fellelhető AUT C++ Példatárból másoltam ki. A feladatokat a Programozás alapjai 2. ZH-ra való felkészülés jegyében válogattam, a példák nagy részét a példatár készítője is „ZH előtt érdemes átnézni" megjegyzéssel látta el.*

*Ihász Dávid*
*I. évf. villamosmérnök BSc*

**Konstansok használata**

*Döntsük el, van-e fordítási idejű hiba az alábbi programrészletekben!*

**a)**
```
const char * str="ZH";
str[0]++;
```

**b)**
```
const char * str="ZH";
str++;
```

**c)**
```
int tomb[ ]={1,2,3};
int* const ptr = tomb;
ptr++;
```

**d)**
```
const int tomb[]={1,2,3};
int* ptr=tomb;
```

**e)**
```
double tomb[ ]={1.0,2.0,3.14};
double* const ptr = tomb;
ptr++;
```

---

Pointerek esetén két érték van: maga a pointer értéke, amely egy memóriacím, illetve a pointer által mutatott érték. Mindkettő lehet konstans, a példák arra kérdeznek rá, hogy a két érték közül melyik.

**a)** Itt a mutatott érték a konstans, vagyis nem módosíthatjuk annak első elemét. Így ez a kódrészlet hibás.

**b)** Ebben a kódrészletben is a mutatott érték a konstans, de a pointer értékét változtatjuk. Vagyis ez nem okoz fordítási idejű hibát.
**c)** Ebben a példában a tömb nem konstans, azonban az első elemére mutató pointer értéke igen. Így a pointer értékének megváltoztatásánál hibaüzenetet kapunk.

**d)** Ezúttal a tömb elemei konstansok. Vagyis ha az első konstans elemére nem konstans pointert próbálunk ráállítani, fordítási idejű hibaüzenetet kapunk: konstansról nem konstansra sosincs automatikus konverzió. Ha a pointer konstansra mutatna, a kódrészlet nem lenne hibás.

**e)** Itt újból a pointer értéke konstans. A konstanst inicializálhatjuk, ezúttal egy tömb elejével. Az inicializáláshoz szükséges konverzió rendben van: a tömb nem konstans egészekből áll, a konstans pointer nem konstans egészre mutat. A konstans pointert ugyanakkor nem növelhetjük meg eggyel, ez fordítási hibát okoz.

**Referencia helytelen/helyes használata**

*Döntsük el, van-e futási idejű hiba az alábbi programrészletekben!*

**a)**
```
int & fv1(int v){
    return v;
}
```

**b)**
```
int fv2(int& rv){
    return rv;
}
```

**c)**
```
int& fv3(){
    int lv=2;
    return lv;
}
```

**d)**
```
int fv3(){
    int lv=2;
    return lv;
}
```

**e)**
```
int& fv2(int& rv){
    return rv;
}
```

**f)**
```
void fv (double & d){
    d+=1;
}
...
int a;
fv(a);
```

**g)**
```
void fv (double & d){
    d+=1;
}
...
int a;
fv((double&)a);
```

A megoldáshoz arra az elvre van szükségünk, **hogy lokális változó/érték szerint átvett argumentum címét nem szolgáltathatjuk ki a függvényen kívülre sem pointeren, sem referencián keresztül**. Ez **a hiba futási időben jön elő**, bár warningot kapunk. (SzC++Ny 2.4 fejezet).

**a)** A kódrészlet érték szerint átadott argumentumot ad vissza referencia szerint, ami súlyos futási idejű hiba (warningot ad a fordító).

**b)** A függvény nem referenciával/pointerrel tér vissza, így biztos nem szolgáltat ki semmilyen címet a függvényen kívülre, ez helyes.

**c)** A kódrészlet lokális változóra ad vissza referenciát, ezért hibás, futási idejű hiba (warningot ad a fordító).

**d)** A függvény nem referenciával/pointerrel tér vissza, így biztos nem szolgáltat ki semmilyen címet a függvényen kívülre, vagyis a kódrészlet referencia szempontjából helyes.

**e)** A referencia szerint átvett paraméter nem lokális, a függvény hívásának hatókörében is létezik, ezért erre akár referenciával/pointerrel is visszatérhetünk, ez referencia szempontjából helyes.

**f)** int és double& között nincs automatikus konverzió, vagyis ez fordítási hiba.

**g)** Ha kierőszakoljuk  konverziót, akkor futási iedjű hibát idézünk elő, mert az int és a double& nem kompatibilis típusok.

**<u>Hibák az osztályokban</u>**

*Keressük meg a hibát az alábbi kódrészletekben!*

**a)**
```
class A{
    int a;
public:
    f()const{
        a++;
    }
};
```

**b)**
```
class A{
    int a;
    static void f(){a++;}
};
```

**c)**
```
class C{
    const int x=2;
};
```

**d)**
```
class A{
    static int c=3;
};
```

**a) Konstans tagfüggvények nem módosíthatják a tagváltozókat**. Két módon javíthatjuk ki:
- a ***mutable*** kulcsszót használjuk, amely pont ezt teszi lehetővé: mutable int a;
- eltávolítjuk a konstanst a tagfüggvény deklarációjából

**b) Statikus függvényből nem érjük el a nem statikus függvényeket és tagváltozókat** (nem megy át a *this* pointer). Vagy a tagváltozót tesszük statikussá, vagy a függvényt nem azzá.

**c)** és **d)** Csak konstans és statikus tagváltozónak adhatunk így kezdeti értéket. **A csak konstans tagváltozót konstruktor inicializáló-listájáról inicializáljuk, a csak statikust külön a definiáláskor** (általában a .cpp-ben).

**<u>Objektumpéldányok száma</u>**

*Írjunk olyan osztályt, ami számolja, hogy hány objektumpéldány létezik belőle és ezt le is lehet tőle kérdezni!*

**A megoldás lényegét a statikus változó azon tulajdonsága adja, hogy osztályszintű, vagyis minden objektumra közös.**

```
#include<iostream>

using namespace std;

class Counter{
public:
    static unsigned int counter;
    Counter(){counter++;}
    ~Counter(){counter--;}
    static unsigned int getCount(){return counter;}
};

// A statikus változót definiálni is kell, static kulcsssszó nélkül: unsigned int Counter::counter=0
```

**Tetszőleges osztály kiírása a standard kimenetre**

*Mutassa be és illusztrállja egy rövid példával, hogyan lehet egy saját osztály (B) számára túlterhelni a << operátort annak érdekében, hogy a C++ szabványos kimenetét használhassuk! Pl: B b; cout << b;*

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

class B {
 int Data;
 // külső függvényként kell megvalósítani, nem pedig tagfüggvényként
 friend ostream &operator<<(ostream &os, const B &b);
};

ostream &operator<<(ostream &os, const B &b) {
 os << b.Data;
 return os;
}
```

**Tetszőleges osztály beolvasása a standard bemenetről**

*Mutassa be és illusztrállja egy rövid példával, hogyan lehet egy saját osztály (A) számára túlterhelni a >> operátort annak érdekében, hogy a C++ szabványos bemenetét használhassuk! Pl: A a; cin >> a;*

```
#include <iostream>
using namespace std;
class A {
 int Data;
 // külső függvényként kell megvalósítani, nem pedig tagfüggvényként
 friend istream &operator>>(istream &is, A &a);
};

istream &operator>>(istream &is, A &a) {
 is >> a.Data;
 return is;
}
```

**Hibakeresés túlterhelt operátorokban**

*Döntsük el, okoznak e hibát az alábbi programrészletek! Ha igen, adjuk meg, hogy fordítási vagy futási idejű-e a hiba, illetve a hiba helyét és magyarázatát! Javasoljunk megoldást, ha van hiba! A programrészletekben elírások nincsenek, az összes meghívott függvény ill. hivatkozott osztály létezik. Ha nincs megadva a kódrészletben, akkor feltételezzük a helyes működést.*

**a)**

```
String::String(const String& theOther){
    ...
        *this=theOther;
}

String & String::operator=(String theOther){
    ...
};
```

**b)**

```
Vector:: Vector (const Vector & theOther){
    ...
    *this=theOther;
}

Vector Vector::operator=(Vector& theOther){
    ...
};
```

**c)**

```
String& String::operator+ (String theOther){
    ...
    return theOther;
}
```

**d)**

```
String& String::operator+ (String theOther){
    String s;
    ...
    return s;
}
```

**e)**

```
Vector::Vector (const Vector theOther){
    ...
    *this=theOther;
}

Vector & Vector::operator=(Vector theOther){
    ...
};
```

**f)**

```
String& String::operator=(String theOther){
     ...
    return theOther;
}
```

**g)**

```
Vector & Vector::operator+ (Vector theOther){
    Vector s;
    ...
    return s;
}
```

**h)**

```
void String::operator=( String theOther){
    ...
    return *this;
};

...

String s1,s2,s3("Santa Claus");
s1=s2=s3;
```

**i)**

```
Complex & Complex::operator− (Complex theOther){
    return theOther;
}
```

---

A megoldáshoz fontos, hogy tudatosítsuk: **a túlterhelt operátorok tulajdonképpen függvények** (leszámítva a speciális kiértékelést híváskor és a speciális szintaxist). Ez azt jelenti, hogy **lokális változó/érték szerint átvett argumentum címét továbbra sem szolgáltathatjuk ki a függvényen kívülre sem pointeren, sem referencián keresztül**. Ez a hiba futási időben jön elő, bár warningot kapunk. (SzC++Ny 2.4 fejezet).

A másik fontos dolog **az = operátor és a másolókonstruktor közötti összefüggés**: ha a másolókonstruktorból az =operátort hívjuk, akkor az = operátorban nem használhatunk érték szerinti átadást, hiszen az a másolókonstruktor további hívását jelentené, amely végtelen ciklust eredményezne (SzC++Ny 6.4 fejezet eleje). Mivel a fordító ezt nem ellenőrzi (warningot itt is ad), ez a hiba is csak futási időben derül ki a verem túlcsordulásával, ami a végtelen rekurzió következménye.

Megjegyezzük még, hogy a hiba többféleképpen kijavítható, bármely megoldás ugyanolyan mértékben elfogadott, az „elegánsabb” megoldások a szokásos megoldásra próbálnak rámutatni.

**a)** Itt az =operátort hívjuk a másolókonstruktorból, vagyis érték szerinti paraméterátadás az operátor=-ben végtelen ciklust okoz. A paraméterlistában pontosan ez történik, amennyiben a paramétert referencia szerint vesszük át, akkor kijavítjuk a hibát.

**b)** Itt a hiba nagyon hasonló az előzőhöz, csak itt a visszatérési értéket kell referencia szerint visszaadnunk.

**c)** Itt egy érték szerint átadott argumentumot adunk vissza referencia szerint. Vagy érték szerint kell visszaadnunk, vagy referencia szerint kell átvennünk. Különösen elegáns, ha konstans referenciaként vesszük át, ez a megoldás a + operátor szokásos használatát is figyelembe veszi. (A mellékhatás a + operátor argumentumain ritkán tekinthető követendő megoldásnak.)

**d)** A kódrészlet egy lokális változót ad vissza referencia szerint. A hiba javításaként érték szerint kell visszaadnunk a lokális változót, és nem referencia szerint.

**e)** A kódrészlet az a) és b) feladat hibáit egyesíti. A hiba javítása is azonos: referencia szerint vesszük át az argumentumot, és referencia szerint térünk vissza.

**f)** A kódrészletben található hiba megegyezik a c) feladatnál tapasztalttal. Az ott bemutatott megoldások helyesek, ugyanakkor mivel az = operátorról van szó, konstans referenciaként vesszük át a paramétert, ez a szokásos megoldás.

**g)** Itt a d) példával analóg a hiba, javítása ugyancsak érték szerinti visszatéréssel.

**h)** Ebben a feladatban nincs benne a másolókonstruktor, vagyis elképzelhető, hogy nem hívja az = operátort. Nem szép megoldás, ne kövessük, de nem is hiba. Viszont az s1=s2=s3 kifejezés átírva:

operator = (s1, operator= (s2, s3))

Ebből az alakból jól látható, hogy az =operátornak szükség van a visszatérési értékére, amely pedig void. A függvényhívás paramétereinek biztosítása a fordító (compiler) feladatkörébe tartozik, ezért ez fordítási hiba. Egy másik fordítási hiba, hogy void visszatérésű függvény nem térhet vissza értékkel. A javítást valamelyest a példa is sugallja: a visszatérési értéket javítsuk String&-ra, és különösen elegáns konstans referenciával visszatérni, bár a példában az érték szerinti paraméterátvétel meglehetősen nehézzé teszi az elegáns végeredményt csak a hibákat javítva.

**i)** A kódrészlet ismét variáció egy témára (c), f)): érték szerint átvett argumentummal térünk vissza referencia szerint. A problémát a szokásos módszerrel orvosolhatjuk, a legelegánsabb a konstans referencia szerinti paraméterátadás.

**Hibák a többszörös öröklésben**

*Mi a hiba az alábbi példákban?*

**a)**

```
class A{
    ...
};

class B{
    ...
};

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

B*pb= new C;
A*pa=(A*)pb;
C* pc = (C*)pv;
delete pc;
```

**b)**

```
class A{
    ...
};

class B{
    ...
};

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

B*pb= new C;
void* pv=pb;
C* pc = (C*)pv;
delete pc;
```

---

**a)** Többszörös öröklésnél sose végezzünk típuskonverziót keresztbe (v.ö. SzC++Ny 7.4), mert elveszítjük a leszármazott eredeti struktúráját! Csak a C bevonásával tehetjük meg:

```
A*pa=(A*)(C*)pb;
```

**b)** A void* típusra való konverzió esetén is elveszíthetjük a leszármazott struktúrájára vonatkozó információt (v.ö. SzC++Ny 7.4.2). A void* elveszti a B-re vonatkozó információt is, ezért azt a konverziót is meg kell adnunk, hogy a fordító helyesen tolja el a pointereket.

```
A*pa=(A*)(C*)(B*)pv;
```

**<u>Kivételkezelési hibák</u>**

*Döntsük el, van-e futási idejű hiba az alábbi programrészletekben!*

**a)**

```
char* str= new char[10];
if(isError()) //Ez a sor OK.
    throw logic_error("Baj van");
delete[] str;
```

**b)**

```
int a;
... // az a változó értékét módosítjuk
int* ptr = new int (32);
if(a==0) throw logic_error("a==0");
```

**c)**

```
class exception{...};
class logic_error: public exception{...};
try{
    ...
}
catch(const exception& e){...}
catch(const logic_error& e){...}
```

**d)**

```
try{
    ...
    throw new logic_error("Problem.");
}

catch(const logic_error* pe){...}{
    cerr << pe->what() << endl;
}
```

---

A kivételkezeléssel kapcsolatos leggyakoribb probléma az **erőforrás-szivárgás**. A lokális változók felszabadításáról a kivételkezelés mechanizmusa automatikusan gondoskodik, viszont a dinamikusan lefoglalt memóriaterületek nem.

Több dolgot tehetünk az ilyen jellegű hibák kiküszöbölésére:

**1.** Egy lokális változóba „csomagoljuk”, ez egy olyan osztály, amely felüldefiniálja a dereferencia „*” operátort (úgy tudjuk használni mint egy pointert), tagváltozóként tartalmazza a pointert, de ugyanakkor a destruktorában felszabadítja a pointert.

Íme egy példa int pointerekre:

```
class int_ptr{
    int*ptr;
public:
    int_ptr(int *ptr):ptr(ptr){}
    ~int_ptr(){delete ptr;}
    int& operator*(){return *ptr;}
    // int esetén elég lenne int-tel visszaterni
    // int& helyett, viszont objektumoknál így érdemes
    const int& operator*()const{return *ptr;} // ezzel konstanst is le tudunk kérdezni
};
```

A használata az alábbi:

```
    int_ptr ptr (new int(3));
    *ptr = 4;
    cout << *ptr;
```

Ha az objektum kimegy a hatókörből, akkor a destruktor felszabadítja. Ugyanez vonatkozik a kivételdobás esetére. Ha a fenti osztályt sablonná alakítjuk, akkor tetszőleges pointerre alkalmazható. Ilyen osztályt nem kell írnunk a Szabványos C++ Könyvtárban az auto_ptr sablon pont erre való (SzC++Ny 10.3.2). Ezt ugyanígy kell használnunk:

```
    auto_ptr<int> ptr(new int(3));
    *ptr = 4;
    cout << *ptr;
```

Ez mindig jó, amikor egy függvényen belül szeretnénk használni egy pointert, például mert nem tudjuk fordítási időben az adat méretét, de azt csak ideiglenesen a függvényben használjuk, a függvényen belül polimorfizmusra van szükségünk.

**2.** Ha nem lokálisan használjuk, akkor pedig tagváltozóként kezeljük, amelynek felszabadításáról az osztály destruktora gondoskodik.

**3.** Természetesen mindig elkaphatjuk az összes kivételt, felszabadíthatjuk a kivétel esetén felszabadítandó erőforrásokat, majd tovább dobjuk a kivételt. Ez elég körülményes megoldás. Ugyanakkor a konstruktorban dobott kivételek esetén csak így tudjuk felszabadítani a tagváltozókat. Félbeszakadt konstruktorhívás esetén nem hívódik meg a destruktor (SzC++Ny 10.3.3), ezért a félbeszakadt konstruktornál fel kell szabadítanunk a pointer-tagváltozókat. Egyéb esetben használjuk a másik két megoldás valamelyikét.

Ezek után nézzük az egyes feladatokat!

**a)** Egyértelmű az erőforrás-szivárgás. Ha a programrészlet kivételt dob, nem hívódik meg a delete[] sor. Többféleképpen javíthatjuk, a kivétel előtt felszabadíthatjuk, vagy auto_ptr-t használunk.

**b)** Ugyanaz a helyzet, mint az a) esetben.

**c)** Ez egy futási idejű hiba. A kivételeket a catch blokkban a legszigorúbbal kell kezdeni, és fokozatosan a legáltalánosabb felé haladva kell felsorolni, különben az általános elkapja a kivételt a speciális elől. Pontosan ez történik a példában: a logic_error rész sohasem hívódik meg.

**d)** Kivétel esetén a lokális kivételváltozó lemásolódik a verem visszacsévélése előtt. Esetünkben ez a pointer, de az általa lefoglalt területet fel kell szabadítanunk, amit nem teszünk meg a kódrészletben. Aranyszabály: sose dobjunk pointert kivételként, és mindig konstans referenciaként kapjuk el.

## Generikus osztályok (template, sablon)

**Követelmények a tárolandó típussal szemben:**

**1.** Legyen alapértelmezett konstruktora - tömb foglalásakor a default konstruktora hívódik meg a típusnak. (**Paraméter nélkül konstruktor**, new-val, tömbben hozzuk létre)

**2.** A generikus osztályban használt **operátorok** (=, <,>, ==, ...) **működjenek helyesen a tárolt típussal**. (Tehát ha pl. saját osztályt akarunk tárolni, akkor ott írjuk meg az operator=, ... operátorokat!)

**3.** Tipp (szintaxis): A globális << operátor függvénysablon lesz, annak friendként való deklarálása:

```
template<class U> friend
std::ostream & operator << (std::ostream& os, const Vector<U>& v);
```

Budapest, 2010.05.