Programozási nyelvek I. C++.

6. előadás

A jegyzetet *Pataki Norbert* előadásán *Lanka Máté* készítette.
Előadás ideje: 2017.03.29. 12:00-14:00.

//Előző EA-n betegség miatt nem voltam, ezért hiányzik az 5. előadás jegyzete.

Előző EA-n láthatóságról és az élettartamról volt szó. A láthatósági dolgok nagyjából végig lettek beszélve, de az élettartamról csak a változókezelésekről volt szó.

A mai alkalommal tehát az élettartam szabályokat fogjuk részletezni. Volt már szó az automatikus változók élettartamáról, de volt szó dinamikus objektumokról is, ezeket ma még átnézzük, lesz még szó lokális statikus változókról is, globális névtérszintű változókról, illetve globális osztályszintűekről is. Lesz még szó osztály adattagokról, tömbelemekről, valamint temporális objektumok élettartamát is megnézzük.

Élettartamnál fontos, hogy amint megszűnik valami, lehet, hogy a memóriában látszódik, de miután megszűnik élettartamilag, semmit nem tehetünk vele.

A dinamikus objektumok, egy picit félreérthetően, a következők:

```
int *p = new int[5];
```

Ezen a ponton, miután az operációs rendszertől futásidőben kérünk memóriát, ott állhat bármilyen adat. Az operációs rendszer annyi memóriát ad, amennyi szükséges, a mi feladatunk, hogy fel legyen ez szabadítva. Van egy pointer a stack-en, ami mutat valahova, oda ahova az operációs rendszer adott. Ennek nincsen neve, névtelen. Amikor dinamikus változóról beszélünk, nem a *p-ről beszélünk. Ez a heap-en jön létre, és nekünk fel kell szabadítanunk, ha már nincs rá szükségünk. Maga a pointer, ha kimegy a scope-ből, nem fogja felszabadítani a tárterületet, ezt a compiler nem is tudja megcsinálni. Egy dinamikus változó élettartama tehát a *delete[]*-ig tart. Ha tehát azt mondom, hogy

```
p = new int(5)  //1 db int
delete p;
```

Ezek után a *p* már csak egy egyelemű tömbre, azaz egyetlen helyre mutat. A korábbi lefoglalt memóriahely felszabadult, és a *p*-t már csak egy int-re mutatjuk. Sajnos, ha itt valami kivétel jött, nem biztos, hogy garantált, hogy eljutok a *delete*-ig, ha nincs lekezelve, hogy kivétel esetén mit csináljon. Ezt előző héten részleteztük, hogy miért jó, ha a dinamikus változókat visszavezetjük az automatikusokra.

Tehát dinamikus objektumok létrejönnek, ha *new*-t elérjük és megszűnnek, ha a *delete*-et érjük el. Viszonylag egyszerű szabály van: kétfajta *new* és *delete* van, ezeket nem keverjük össze, mert nem definiált viselkedés jön létre. Ha meg elfelejtük törölni, akkor az már memóriaszivárgáshoz vezethet.

Nézzük a **lokális statikus**okat. Nézzünk egy egyszerű példát:

```
void f()
{
        static int v[] = {3,8,5};
}
```

Akkor jön létre ez a lokális statikus tömb, amikor elérjük ezt a deklarációt. Viszont nem szűnik meg, mint egy normális lokális változó, hanem egészen a program futásának végéig a memóriában marad, akárhányszor eljutok odáig, és akárhányszor hivatkozok a tömbre, az a legelső alkalommal

jön létre, és onnantól kezdve végig azzal foglalkozunk. Csak a program futásának végén szűnnek tehát meg.

Vannak **globális névtérbeli**ek, pl. az *std::cout* egy ilyen. A **globális osztály statikus tag**ok nagyon egyszerű szabállyal rendelkeznek: program indulásakor létrejönnek, program végén pedig megszűnnek. Ezeket mindenhonnan el tudjuk érni, ha statikus, akkor csak egy fájlon belül. A kettő között ott a különbség, hogy míg a lokális statikusoknál függ az, hogy eljutunk-e oda, meghívjuk-e az adott függvényt, addig utóbbinál nem.

Osztály adattagok. Viszonylag egyszerű a szabály. Létrejönnek akkor, ha a tartalmazó objektum létrejön és megszűnnek, ha a tartalmazó objektum is megszűnik. Ez mit jelent és miért érdekes ez a szabály, úgy általában? Vegyünk egy egyszerű példát:

```
class Person()
{
	public:
		std::string name;
}
void f()
{
	Person p;
	…
}
```

Tehát ahol létrehozom ezt a person-t, akkor létre is kellett hozni az adattagot. Létre kell jönnie ennek a p.name-nek is, ami nem lehet null. Ez egy megkonstruálható adattag, aminek meg lehet tudni a méretét, ami nyilván 0 lesz, mert üres lesz a string, azaz működik akármi is. A lényeg, hogy a p.name-nek létre kell jönnie. Viszont ha az *f* függvény megszűnik, akkor az a *name* is megszűnik. Dinamikus élettartammal a *heap*-re kerülnek, fel fog szabadulni az a memóriahely, visszaadjuk a memóriának.

Hasonlóan viselkednek még a tömbelemek is. Ezek létrejönnek akkor, ha maga a tartalmazó tömb is létrejön, illetve megszűnnek akkor, ha a tartalmazó tömb is megszűnik. Ha azt akarjuk, hogy Person-ok tömbje jöjjön létre az alábbi módon:

```
void f()
{
	Person v[6];
}
```

akkor ebben a 6 elemű tömbben megkonstruált Person-öknek kell létrejönniük. Hat darab megkonstruált Person-nak kell ottlennie, megkonstruált adattagokkal is. Nincs olyan állapota egy objektumnak, hogy nem létezik még, ezeknek muszáj. Pointereknél van olyan, hogy még egy ideig nullpointer, de maga a pointer egy leallokált tárterületen van.

Még egy szabály, ami hátravan, a **temporális**ok. Sajnos időnként megtéveszti az embert a matematikai szemléletmódja. Miért, mik ezek, hol jönnek létre? C++-ban tudunk pl. írni egy mátrix osztályt. Tökmindegy, ezt milyen reprezentációval oldjuk meg, ez mellékes. Azt tételezzük fel, hogy a mátrixnak van összeadó operátora, de ez esetben ez hogy van megírva? Nyilván vár egy jobb oldali és egy bal oldali mátrix operandust. Azaz ha írunk egy összeadó operátort, hogy meg tudom írni egy operandusra, és két operandusra is. Összeadom, visszaadom, jó. De mi van, ha össze akarunk adni mátrixokat? Nézzük meg:

```
class Matrix
{
```

…
}
Matrix m,a,b,c,d;
…
m = a+b+c+d;

Megbeszéltük, hogy az összeadás balról-jobbra dolgozik, először összeadja az *a*-t és *b*-t, lesz egy temporális mátrixot. Ezt összeadjuk a *c*-vel, kapunk egy újabb temporális mátrixot. Ehhez megint hozzáadjuk a *d*-t, megint kapunk egy temporálist, de ezt értékül adjuk az *m*-nek.

Vegyünk egy másik példát:

std::string s,a,b;
…
consts char *p = (s+a+b);

A stringösszeadás is kétoperandusú művelet ez esetben. Ennek le lehet kérni a *.c_str()* reprezentációját. De mi van akkor, ha én ezt nekiállok kiíratni?

std::cout<<p;

Az a gond, hogy van egy temporális objektumom, azaz kérek egy első elemre mutató pointert. Lefut a string destruktora, kipucolja a tárterületről őket. Ez egy nem definiált viselkedés. Érdemes megnézni, hogy ezek mikor jönnek létre? Akkor, ha a részkifejezés kiértékelése sorra kerül. Megszűnnek viszont, ha a teljes kifejezés kiértékelődik. Felszabadítja maga mögött a leallokált tárterületet.

Nézzünk erre egy **feladat**ot. Egy függvényt fogunk írni, ami paraméterül kap egy kérdést, ezt kiírjuk a felhasználónak, elvárjuk, hogy válaszoljon, és ezt adjuk vissza a függvénynek, amit ezt visszaad. Pl. „Hogy vagy?”, ezek *char* alapú tömbként kerül át a függvénynek, pontosabban pointerként.

Nézzük, ezt viszont hogy lehet megvalósítani?

```
char* answer(const char* q)
{
	std::cout<<q;
	char ans[80];  //Ennél többet csak nem ír le a felhasználó.
	std::cin>>ans;
	return ans;
}
```

Nyilvánvaló, hogy ez a kód nem jó! Milyen hibák vannak benne? Itt ez az egyszerű kód, ami tele van hibákkal. A problémát az okozza, hogy ott az a lokális változó, a tömb, amelyet visszaadunk, és a deklarálódó blokk végén megszűnik. Fontos szabály C++-ban, hogy lokális változóra mutató referenciát nem adhatunk vissza, mert megszűnik. Pointer visszatérési értéknek van értelme. A másik, ami nem jó, az a beolvasás. Mivel ott is első elemre mutató pointerré konvertálódik a tömb. Ezen a tárterületen túl lehet mutatni, ha több mint 80 karaktert akar az ember a tömbbe írni. Ha ezt átírjuk arra, hogy

std::cin.getline(ans,80);

Akkor ez figyeli, hogy maximum 80 karaktert lehessen beírni. Viszont a problémát még mindig nem oldottuk meg. Ha viszont ezt globális változóként oldjuk meg (mármint az *ans*-ot), akkor túléli a függvényhívást.

```
char ans[80];
char* answer(const char* q)
{
	std::cout<<q;
	std::cin.getline(ans,80);
	return ans;
}
```

Mi van, ha azt mondom, hogy írjuk ki a standard outputra, hogy *std::cout<<answer(„Hogy vagy?”);* Ezzel az utolsó megoldással viszont az a baj, hogy ha több szálon fut a kód, akkor is egy tárterületen van rajta.

```
std::cout<<answer(„Hogy vagy?”)<<answer(„Biztos?”);
```

(Azon ne akadjunk fel, hogy a két függvényhívás között épp melyik fut le előbb. De nézzük meg a probléma élettartamra vonatkozó problémáját.

De nézzük meg ezt. A program megkérdezi, hogy van a felhasználó, visszaírja, hogy „~~fenébe,~~ katalógusos az előadás”, visszakérdez, hogy biztos, válasz igen. A tárterületen viszont csak egy válasz lesz, nem lesz minden egyes válasznak külön tárterülete. Itt az a gond, hogy ha több aktív answer() függvényem van, a későbbiek felülírják a korábbiakat. Ráadásul volt szó arról, hogy nem szeretjük a globális változókat, mert azokat mindenki ész nélkül át tudja írni ráadásul. Egy fokkal szerencsésebb, ha annyit mondunk, hogy:

```
char* answer(const char* q)
{
	std::cout<<q;
	static char ans[80];
	std::cin.getline(ans,80);
	return ans;
}
```

Ez a lokális statikus tömb is olyan, hogy ha több aktív példánya van, akkor is egy tárterületen osztozkodnak. Csak annyit javítottunk, hogy az ans láthatóságát szűkítettünk.

Próbáltunk lokális változóval, globális változóval, lokális statikussal. Egyik se jött be. ☹ Következő megoldás, hogy próbáljuk dinamikusan.

```
char* answer(const char* q)
{
	std::cout<<q;
	char* ans = new char[80];
	std::cin.getline(ans,80);
	return ans;
}
```

Megint remek, mert a new-val leallokált tárterület túléli a függvényhívást, és visszaadhatom a pointert. Mi mégis a probléma? A hívónak van lehetősége, hogy lefoglalja a tárterületet, és szükség esetén felszabadítsa. Ezzel nem lehet mit csinálni, nem szimmetrikus a megoldás, nem lehet semmivel se kikényszeríteni, hogy a *delete* ezt törölje. Van rá lehetőség, hogy a hívó felszabadítsa, de nem garantált, hogy ez meg is történik.

```
char* a = answer(„Hogy vagy?”);
delete[] a;
```

Van rá lehetőség, de nem látszik, hogy ez meg is fog történni. Ha kimarad, akkor viszont memóriaszivárgás történik. Az lenne a logikus elképzelés, hogy azt mondja az answer() hívója, hogy ha ő allokálja a tárterületet, akkor ő is szabadítsa is fel. Azaz ne csak 1 paraméter legyen, hogy mit kellene megkérdezni, hanem adja meg a válasz területét is.

Adjunk vissza egy std::string-et. Ha azt adjuk vissza, hogy *std::string anser(const char* q),* az érdekel, hogy minket miért motivál a string használata?

```
std::string anser(const char* q)
{
	std::cout<<q;
	std::string ans;
	std::getline(std::cin,ans);
	return ans;
}
```

Amikor visszaadjuk a stringet, az adott tárterületen lévő adatot kimásoljuk. Ezután fog csak kitörlődni a string a memóriaterületről, felszabadul a hely, viszont vissza tudjuk adni a tartalmát, és csak utána felszabadul a tárterület, nincs memóriaszivárgás. Ez volt tehát a motivációnk. A string létrehozásakor lefoglaljuk tehát a tárterületet, a return-nél kimásoljuk onnan, és a végén pedig ki is töröljük. Később persze ez a másolgatás egy bizonyos ponton hátrány, egy bizonyos ponton pedig előny.

Deklaráció, definíció

Általában fordítási egységekben gondolkodunk, ezek külön-külön fordulnak le. Az a.cpp-ből lesz a.obj, a b.cpp-ből b.obj, ezek egymástól külön-külön kell lefordulniuk, önállóan, még akkor is, ha hivatkoznak a másikra. Nagyon gyakori, hogy ilyenkor különböző függvényeket, típusokat, akár változókat is megoszthatunk különböző fordítási egységek között. Ha van az a.cpp-ben egy globális integer változóm, azt el tudom érni a b.cpp-ből is. Ott használhatom az *extern int x*-et is, a kettő *x* ugyanaz lesz.

Van a C++-ben egy olyan, hogy *One Definition rule*. Azaz egy helyen kell csak definiáltnak lennie, ha pl. az *x*-et máshol is akarom használni, azt deklarálhatom bármennyiszer, de egy fordítási egységnek tartalmaznia kell a definíciót, benne kell lennie a tárterületben, amit a többi is tud használni. Van egy linkelési lépés a fordítás után, ahol ezeket meg tudjuk találni a fordítási egységben.

Hasonlóképp, ha van egy függvényünk, amelyet több fordítási egységből akarunk használni, pl van egy f() függvényem, amely növeli az x-et mindig, akkor ezt is egy helyen meg kell írnom. Ha azt mondom, hogy szeretném ezt használni, figyelnem kell, hogy helyes paraméterezéssel használom. Valamennyi minimális információra szüksége van a compiler-nek is. Legyen egy *d.cpp*, ahol definiálnom kell ezt: *void f();*, azaz paraméter és visszatérési érték nélküli. Viszont nem mindegy, hogy a fordításhoz képest ezek a felosztások mikor történnek meg.

A C++ bevezette az inline függvény fogalmát. C-ben előszeretettel használták azokat a makrókat, hogy:

```
#define SQ(x) ((x)*(x))
```

Ha viszont C++-ban gondolkodunk, akkor használjuk az inline függvényt:

```
inline double sq(double x)
{
	return x*x;
}
```

Kettő között az az alapvető filozofiai különbség, hogy míg előbbi szövegátalakítóval feldolgozott buta megoldás, hogy ne kelljen a függvényhívás költségét „kifizetni”, míg utóbbinál, a C++ azt mondta, hogy ne a preprocesszortól függjön ez, hanem csinált egy nyelvi eszközt, nem akarunk belőle object-szintű nyelvi kódot, hanem ahol ezt meghívják, oda ezt helyettesítsék be. Ha ezt több fordítási egység között akarom megosztani, akkor ebből nem lesz object kód, mert ahol ezt meg tudjuk hívni ezt a függvényt, oda behelyettesítsük a törzsét. Épp ezért ezt nem a linkerrel duplikáljuk le, hanem a preprocesszorban. Ha több fordítási egységben akarom használni, akkor ezt header-ben kell megírnunk.

Hasonló ezekhez, hogy azt mondom, hogy

```
class complex
{
        double re, im;
        …
};
```

Ez megint egy definíció, ebből is lehetnek problémák, erre jók az include guard-ok, amelyekről volt már szó.

A class complex; tehát egy deklaráció. Ilyenkor nem tudok létrehozni egy komplex szám objektumot. De egy pointert igen (*complex *p = 0)*.

Van még egy érdekes helyzet. A definíciók általában deklarációk is, viszont van, amikor nem esik egybe a kettő. Ha van egy osztályunk:

```
class Foo
{
        static int cnt;
        ….
};
```

Ha kifejtem ennek a műveleteit a foo.cpp-ben, akkor ezt a globális névtérszintű változót be akarom vezetni, és definiálni is. Ha azt mondom, hogy *int Foo:cnt;*, erről ezt ponton el kell mondani, hogy jöjjön létre, az utóbbi tehát egy definíció, a fenti példában pedig deklaráció, viszont mindkettőre szükség van.