Dinamikus adattagok. Copy constructor & assignment. Destruktor

OOP és DB - OOP 3. óra

Dinamikus adattagok

A C++-ban *dinamikusnak* nevezünk minden olyan adatot, ami nem a *stack*-en, nem az *adatszegmens*-en, hanem a *heap*-en (magyarul: halmon) kerül allokálásra

A *stack*-re kerülnek a lokális változók, mérete nő és csökken a függvényhívásokkal (ha egy függvény visszatér, a lokális változók felszabadulnak a stacken).

Az *adatszegmens*-re kerülnek a globális illetve statikus változók.

Dinamikus adattagok

A *heap*-et / *halmot* akkor használjuk, amikor:

szeretnénk, ha az adat “túlélné” a függvények visszatérését, ugyanakkor mégsem szeretnénk, hogy globális legyen és / vagy mérete változni tud

pl. egy láncolt listáról nem tudhatjuk előre, hogy hány eleme lesz - elemeit a stacken nem is lenne értelme lefoglalni

(persze maga a láncolt lista, mint változó, lehet a stacken, de az általa kezelt adatok dinamikusak)

Dinamikus adattagok létrehozása / felszabadítása

Dinamikus változót a *new* kulcsszóval tudunk létrehozni. Ez egy pointert ad vissza a megfelelő memóriacímre.

Amit *new*-val létrehoztunk, egyszer *delete*-elni is kell. Osztályon belül ez a *destruktor* feladata!

Ha a programunk sokáig fut és nem szabadítjuk fel a dinamikus mem.területet, tele lesz szeméttel, lassabban fog futni és idővel le is állíthatja az operációs rendszer a programot. Nemcsak hogy nem elegáns, de komoly gondokat is okozhat, ha a delete-ről elfelejtkezünk.

Ha a *delete*-et 2x hívjuk meg, az sem jó, mert időközben már a futtatási környezet lehet, hogy ugyanazt a mem.területet már másra használja

Dinamikus adattagok - példa (dinamikus láncolt lista)

Először vegyük a Node osztályt. Ez egy beágyazott osztály is lehetne, de olyat még nem vettünk.

```
class Node {
    int value;
    Node* next;
public:
    Node(int val) : value(val), next(nullptr) {}
    int getValue() const { return value; }
    void setValue(int x) { value = x; }
    Node* getNext() { return next; }
    void setNext(Node* n) { next = n; }
};
```

Dinamikus adattagok - példa (dinamikus láncolt lista)

A LinkedList osztály dinamikusan hoz létre Node objektumokat a heap-en:

```
class LinkedList {
    Node* root;
public:
    LinkedList(int firstValue) : root(new Node(firstValue)) {}
    void addNode(int val) {
        Node* last = root;
        while (last->getNext() != nullptr) {
            last = last->getNext();
        }
        last->setNext(new Node(val));
    }
```

Dinamikus adattagok - példa (dinamikus láncolt lista)

Amit dinamikusan létrehoztunk, azt egyszer törölni is kell. Erre való a destruktor, ami ugyanolyan nevű, mint az osztály, csak van előtt egy tilde (~) és nincs visszatérési típusa

```
class LinkedList {
    Node* root;
public:
    LinkedList(int firstValue) : root(new Node(firstValue)) {}
    ~LinkedList() { // ez itt a destruktor!
        Node* nextNodeToDelete = root;
        while (nextNodeToDelete != nullptr) {
            Node* next = nextNodeToDelete->getNext();
            std::cout << "Deleting node w/ value " << nextNodeToDelete->getValue() <<
                std::endl;
            delete nextNodeToDelete;
            nextNodeToDelete = next;
        }
    }
```

Dinamikus adattagok - példa (dinamikus láncolt lista)

Let’s have some fun...

```
class LinkedList {
    Node* root;
public:
    LinkedList(int firstValue) : root(new Node(firstValue)) {}
    void print() {
        Node* current = root;
        while (true) {
            Node* next = current->getNext();
            if (next == nullptr) {
                std::cout << current->getValue() << std::endl;
                break;
            }
            else {
                std::cout << current->getValue() << ", ";
                current = next;
            }
        }
    }
```

Dinamikus adattagok - példa (dinamikus láncolt lista)

Let’s have some fun...

```
void f() {
    LinkedList my_ll(5);
    my_ll.addNode(6);
    my_ll.addNode(7);
    my_ll.addNode(8);
    my_ll.print();
    std::cout << "about to destruct my_ll" << std::endl;
}

int main()
{
    f(); // kulon fv-be tesszuk, hogy a destruktort teszteljuk
```

```
5, 6, 7, 8
about to destruct my_ll
Deleting node w/ value 5
Deleting node w/ value 6
Deleting node w/ value 7
Deleting node w/ value 8
```

Összefoglaló

Most már akkor látjuk, hogy többféle konstruktor létezhet (default konstruktor - aminek üres az argumentum listája - további tetszőleges konstruktorok), valamint hogy létezik olyan, hogy destruktor.

Destruktort muszáj készítenünk, ha az osztálynak van dinamikusan allokált adata.

Ha egy osztály neve az, hogy *Something*, akkor:

A default konstruktor szignatúrája: *Something()*

A destruktor szignatúrája: *~Something()*

Copy constructor

Említettük, hogy ha semmilyen konstruktort nem hozunk létre, a fordító automatikusan generál nekünk egy default constructor-t. Ennek legtöbb gyakorlati esetben nincs sok haszna, ezért érdemes mindig saját konstruktort készíteni.

Ugyanígy - alapesetben - a fordító létrehoz nekünk egy copy konstruktort is. Ez lehetővé teszi, hogy adott típusú objektumot egy másik, már létező, ugyanolyan típusú objektum alapján létrehozzunk.

Copy constructor

Például:

```
5, 6, 7, 8
5, 6, 7, 8
about to destruct my_ll and my_second_ll
```

```
void f() {
    LinkedList my_ll(5);
    my_ll.addNode(6);
    my_ll.addNode(7);
    my_ll.addNode(8);
    LinkedList my_second_ll(my_ll); // mukodik alapbol is!
    my_ll.print();
    my_second_ll.print();
    std::cout << "about to destruct my_ll and my_second_ll" << std::endl;
    std::cin.get();
}

int main()
{
    f(); // kulon fv-be tesszuk, hogy a destruktort teszteljuk
```

Copy constructor

Itt csak annyi a probléma, hogy az automatikusan generált copy constructor nem olyan okos. Nem foglal le pl. új dinamikus memória-területet, ehelyett egyszerűen felhasználja ugyanazt, amit my_ll objektum már használ.

Éppen ezért, a Visual Studio futtatókörnyezete csúnya crasht produkál, amikor my_ll vagy my_second_ll változók közül az egyiket a destruktorral felszabadítja, mivel érzékeli, hogy ezzel a másik változó memóriaterületén garázdálkodtunk...

```
5, 6, 7, 8
5, 6, 7, 8
about to destruct my_ll and my_second_ll
```

Copy constructor

Something osztály copy construktorának szignatúrája *Something(Something&)* vagy *Something(const Something&)*.

Például így már jó:

```
5, 6, 7, 8
5, 6, 7, 8
about to destruct my_ll and my_second_ll
Deleting node w/ value 5
Deleting node w/ value 6
Deleting node w/ value 7
Deleting node w/ value 8
Deleting node w/ value 5
Deleting node w/ value 6
Deleting node w/ value 7
Deleting node w/ value 8
```

```
class LinkedList {
    Node* root;
public:
    LinkedList(int firstValue) : root(new Node(firstValue)) {}
    LinkedList(const LinkedList& other) {
        root = new Node(other.root->getValue());
        Node* mycurrent = root;
        Node* othercurrent = other.root;
        Node* othernext = othercurrent->getNext();
        while (othernext != nullptr) {
            mycurrent->setNext(new Node(othernext->getValue()));
            mycurrent = mycurrent->getNext();
            othercurrent = othernext;
            othernext = othercurrent->getNext();
        }
    }
```

Copy assignment

Hasonló, de picit más elven működik a copy assignment (operator=).

Ha egy osztály *b* példányát egyenlővé tesszük *a* példányával (a = b), akkor az *a*-ban levő adatokat kicseréljük *b*-ben levő adatokra

> Ami lényeges, hogy *a* objektum ekkor már létezik! Vagyis destruálni kell azt a mem.területet és egyéb erőforrásokat, amiket lefoglalt
>
> Copy constructor ehhez képest egy vadiúj objektumot csinál

Egyébként a fordító alapból legenerálja a copy assignmentet is (ugyanúgy legtöbbször hibásan - nem szabadít fel memóriát és nem foglal le új memóriát)

Copy assignment

Something osztály copy assignmentjének szignatúrája *Something& operator=(Something&)* vagy *Something& operator=(const Something&)*.

A copy assignment visszatérési értéke: *this

```
LinkedList& LinkedList::operator=(const LinkedList& other) {
    // ez nem konstruktor, hanem assignment, ertekadas
    // tehat: ez az objektum mar letezik!!
    // -> mar lehet lefoglalt mem.terulete, amit fel kell szabaditani
    // lehet ugy is, hogy elkezdjuk felulvagni - es ha maradt valami, azt toroljuk
    // de most inkabb a naiv implementaciot kovessuk!

    // destruktor kodja egy-az-egyben
    Node* nextNodeToDelete = root;
    while (nextNodeToDelete != nullptr) {
        Node* next = nextNodeToDelete->getNext();
        delete nextNodeToDelete;
        nextNodeToDelete = next;
    }
    // DE: hogy ne legyen inkonzisztens allapot, nullptr beallitasa kell!!
    root = nullptr;

    // ezutan jon a tobbi ugyanugy...
```

“Rule of 3”

A híres “Rule of 3” azt mondja ki, hogy amennyiben egy osztályhoz definiálunk saját destruktort, VAGY saját copy constructort, VAGY saját copy assignmentet, akkor a másik kettőt is célszerű megvalósítani

Általános megfigyelés: ezek a metódusok kéz a kézben járnak. Ha az egyiket implementáljuk, az azt sejteti, hogy az osztály nemtriviális dolgokat csinál a memória- (vagy egyéb erőforrás-)foglalások terén. Ilyenkor nem triviális a másolás, nem triviális az értékadás és a felszabadítás sem.

Ha sietünk, lehet olyat is, hogy egyszerűen megtiltjuk a copy-t és assignmentet. Ilyenkor a deklaráció:

```
LinkedList(const LinkedList&) = delete;
LinkedList& operator=(const LinkedList&) = delete;
```