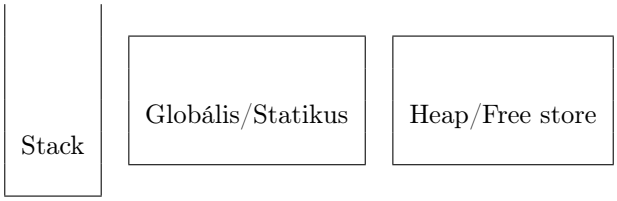
C++
Gyakorlat jegyzet
5. óra

A jegyzetet Umann Kristóf készítette Horváth Gábor gyakorlata alapján. (2018. április 30.)

1. A C++ memóriamodellje

A C++ szabvány több memóriatípust különít el. Ezek közül elsősorban a stack-et használtuk eddig.



1.1. Stack

A második gyakorlaton már volt szó részletesebben a stack működéséről. Jusson eszünkbe egy nagyon fontos tulajdonsága: a blockok végén a változók automatikusan megsemmisülnek. Ebben az esetben nem a programozó feladata a memória felszabadítása.

A stack-en létrehozott változókat szokás **automatikus változók**nak (*automatic variable*) is hívni.

Tekintsük a második gyakorlatról már ismerős kódrészletet.

```
#include <iostream>

int f()
{
    int x = 0;
    ++x;
    return x;
}

int main()
{
  for(int i = 0; i < 5; ++i)
      std::cout << f() << std::endl;
}
```

Kimenet: `1 1 1 1 1`

A stacken lérehozott változók kezelése nagyon kényelmes, mert jól látható, mikor jönnek jönnek létre, mikor semmisülnek meg. Azonban előfordulhat, hogy nem szeretnénk, hogy az `x` változó élettartama megszűnjön a block végén. Ilyenkor egy lehetőség a statikus változók használata.

1.2. Globális/statikus tárhely

Írjuk át a fenti `f` függvényt, hogy `x` ne automatikus, hanem **statikus változó** (*static variable*) legyen!

```
int f()
{
    static int x = 0;
    ++x;
    return x;
}

int main() {/*...*/}
```

Kimenet: `1 2 3 4 5`

Ebben az esetben azonban a függvény első hivásától a program futásának végéig benne marad a memóriában az `x`, így mindig egyre nagyobb számokat ad majd `f()` vissza. Az `x` inicializációja egyszer történik meg, a függvény első hívásakor.

1.2.1. Megjegyzés. Nem szeretjük a `static` változókat. Például a több szálú programok esetén különösen kerülendő ez a programozási stílus.

A globális változók és a statikus változók a memória ugyanazon területén jönnek létre. Ezért hívjuk ezt a területet globális/statikus tárhelynek.

Amennyiben azt szeretnénk, hogy `x` ne semmisüljön meg a block végén, de ne is maradjon a program futásának a végéig a memóriában, arra is van lehetőség. Ebben az esetben viszont a programozó felel a memória kezeléséért.

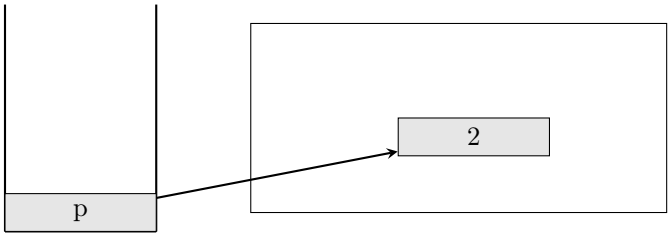
1.3. Heap/Free store

A heapen létrehzoott változókat **dinamikus változók**nak (*dynamic variable*) is szokás szokás hívni. A heap segítségével nagy szabadságra tehetünk szert, azonban ez a szabadság nagy felelősséggel is jár.

```
int main()
{
    int *p = new int(2);
    delete p;
}
```

Fentebb láthatjuk hogyan lehet egy intnek lefoglalni helyet a heapen. Fontos, hogy a stack-et nem kerültük meg, mert szükségünk van egy pointerre, mely a heap-en lefoglalt címre mutat (`p`).

A mutató által mutatott területet a `delete` operátorral tudjuk felszabadítani.



1. ábra. Példa a heap működésére: `p` egy, a heapen lefoglalt memóriacímre mutat.

A heapen nincs a lefoglalt területnek nevük, így mindig szükségünk lesz egy mutatóra, hogy tudjunk rá hivatkozni. **Ha egyszer lefoglalunk valamit a heap-en, gondoskodni kell arról, hogy felszabadítsuk.** Az egyik leggyakoribb hiba a dinamikus memóriakezelésnél, ha a memóriát nem szabadítjuk fel. Ilyenkor a lefoglalt memóriaterületre hivatkozni már nem tudunk de lefoglalva marad: elszivárog (*memory leak*).

Bár az operációs rendszer megpróbál minden, a program által lefoglalt memóriát felszabadítani a futás befejeztével, de nem mindenható. Előfordulhat, hogy egyes platformokon újraindításig nem szabadul fel a memória. Emellett ameddig a program fut, több memóriát fog használni, mint amennyire szüksége van. Ez növelheti a szerverpark költségeit vagy ronthatja a felhasználói élményt.

A dinamikusan lefoglalt memória szabályos felszabadítását számos dolog nehezíti. Fényes példa erre a kivételkezelés, melynél hamarabb megszakadhat a függvény végrehajtása mintsem, hogy felszabadítson minden memóriát. Előfordulhat, hogy egy memóriaterületet kétszer szabadítunk fel, ami nem definiált viselkedés.

Előfordulhat, hogy egy már felszabadított memóriaterületre akarunk írni vagy onnan olvasni. Sajnos ilyen jellegű hibát könnyű véteni, hisz a `delete` a `p` által mutatott memóriaterületet, nem a `p`-t fogja törölni. A `p` továbbra is használható.

1.3.1. Megjegyzés. A nullpointer törlésekor nem történik semmi (*no-op*).

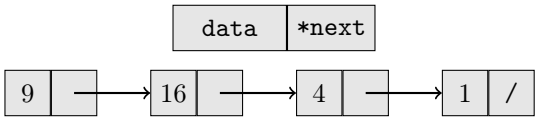
1.3.2. Megjegyzés. Amint elvesztettük az utolsó mutatót, ami egy adott lefoglalt memóriacímre mutat, az garantáltan elszivárgott memória. A szabvány nem foglal magában semmilyen lehetőséget ezeknek a visszaszerzésére.

Láthatjuk, hogy a heap használata hibalehetőségekkel teli, ráadásul az allokálás (memória lefoglalás) még lassabb is, mintha a stack-et használnánk. De miért használjuk mégis? Ha meg lehet oldani, hogy a stack-en tudjunk tárolni valamit, tegyük azt. A stacken azonban véges, hamar be tud telni (*stack overflow*), illetve kötött a változók élettartama. A heap-en e téren sokkal nagyobb a szabadságunk.

2. Osztályok felépítése

A következő pár gyakorlaton egy láncolt listát fogunk implementálni, mely jól demonstrálja majd a dinamikus memóriakezelés veszélyeit is.

A láncolt lista nevéből eredendően nem tömbszerűen (egymás melletti memóriacímeken) tárolja az objektumokat, hanem egymástól független memóriacímeken. Ezt úgy oldja meg, hogy minden adathoz rendel egy pointert is, mellyel a következő listalemet el lehet érni. A lista utolsó elemében a pointer a rákövetkező elem memóriacíme helyett nulpointer értéket vesz fel.



2. ábra. A fenti képen egy listaelem, a lentin egy adatként `int`-et tároló 4 elemű láncolt lista. Figyeljük meg, hogy az utolsó elem pointere nullpointer.

2.1. Struct-ok

Egy láncolt lista elemét implementálhatjuk pl. így:

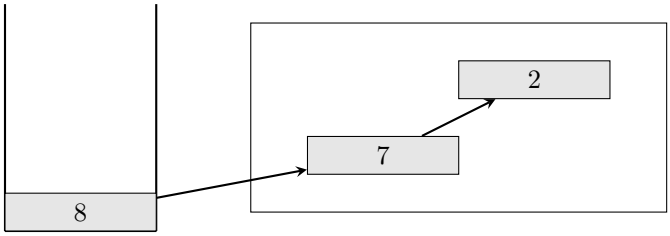
```
struct List
{
    int data;
    List *next;
};
```

Alkalmazzuk is ezt úgy, hogy a listaelemek dinamikusan legyen eltárolva!

```
int main()
{
    List *head = new List;
    head->data = 8; //(*head).data == head->data
    head->next = new List;

    head->next->data = 7;
    head->next->next = new List;

    head->next->next->data = 2;
    head->next->next->next = NULL;

    delete head;
    delete head->next;
    delete head->next->next;
}
```

Ezen a ponton mondhatnánk, hogy készen vagyunk, hisz `List` használható láncolt listaként (bár valójában igen kényelmetlen).

Sajnos a törlést rossz: először törljük a fejelemet (mely az első elemre mutat), viszont az első elem segítségével tudnánk a többi elemet elérni, így mikor a második listaelemet törölnénk, `head` már egy



3. ábra. Láncolt lista. Az első elem stacken, a többi heapen van tárolva.

felszabadított memóriaterületre mutat. Ezt törlés utáni használatnak (*use after delete*) szokás nevezni és nem definiált viselkedés.

A megoldás:

```
delete head->next->next;
delete head->next;
delete head;
```

2.1.1. Megjegyzés. A heap-en arra is figyelni kell, hogy jó sorrendben szabadítsuk fel a memóriát. Ha rossz sorrendben szabadítjuk fel az objektumokat, könnyen a fentihez hasonló hibát vagy memória szivárgást okozhatunk.

A változók a stacken a létrehozás sorrendjéhez képest fordított sorrendben semmisülnek meg, pont emiatt.

Ez a „láncolt lista” eddig elég szegényes. A fő gond az, hogy nagyon sokat kell írni a használatához. Ez sért egy programozási elvet, a DRY-t: *Dont Repeat Yourself*. Itt sokszor írjuk le közel ugyanazt – erre kell, hogy legyen egy egyszerűbb megoldás. Írjunk függvényt az új listaelem létrehozásához!

```
List *add(List *head, int data)
{
    if (head == 0)
    {
        List *ret = new List;
        ret->data = data;
        ret->next = 0;
        return ret;
    }
    head->next = add(head->next, data);
    return head;
}
```

Ez egy olyan rekurzív függvény, mely addig hívja saját magát, míg a paraméterként kapott lista végére nem ér (azaz a `head` egy nullpointer). Amikor oda elér, létrehoz egy új listaelemet és azt visszaadja. A rekurzió felszálló ágában a lista megfelelő elemeit összekapcsolja.

Írjunk egy függvényt a lista által birtokolt memória felszabadítására is.

```
void free(List *head)
{
    if (head == 0)
        return;
    free(head->next);
    delete head;
}
```

Itt a rekurzió szintén a lista végéig megy. A rekurzió felszálló ágában történik a listaelemek felszabadítása. Ennek az oka, hogy a felszabadítás a megfelelő sorrendben történjen meg.

2.1.2. Megjegyzés. A rekurzív függvények nem olyan hatékonyak, mint az iterativ (pl. `for` vagy `while` ciklus) társaik. Továbbá a sok függvényhívás könnyen stack overflow-hoz vezetnek. Azonban jó agytornák, és segíthetnek az alapötletben. Egy rekurzív függvényt mindig át lehet írni iteratívvá.

Beszéljünk arról, mennyi a teher a felhasználón. Eddig tudnia kellett, milyen sorrendben kell felszabadítani az elemeket a listán belüle, de most már elég arra figylenie, hog lista használata után meghívja a `free` függvényt. A felhasználó így kisebb eséllyel követ el hibát, több energiája marad arra, hogy az előtte álló problémát megoldja. Legyenek a függvényeink és osztályaink olyanok, hogy **könnyű legyen őket jól használni, és nehéz legyen rosszul**.

2.2. Osztályra statikus változók

Teszteljünk!

```
int main()
{
    List *head = 0;
    head = add(head, 8);
    head = add(head, 7);
    head = add(head, 2);

    free(head);
}
```

A program lefordult, és a gyakorlaton tökéletesen le is futott. Azonban, ha történt memory leak vagy double free, esetleg use after free, az nem definiált viselkedés. Ezért nem lehetünk benne biztosak, hogy valóban nem történt memóriakezeléssel kapcsolatos hiba. A sanitizerek segítségével megyőződhetünk róla, hogy nem követtünk el ilyen jellegű hibát.

Az osztályon belül statikusként deklarált változókat osztályszintű változóknak is hívjuk, ugyanis minden, az osztályhoz tartozó objektum ugyanazon a statikus változón „osztozkodik”. Ha az egyiken keresztül azt a változót módosítjuk, a többiben módosulni fog. Élettartamuk és láthatóságuk a program elejétől végéig tart.

Hozzunk létre `List`-ben egy számlálót, ami számon tartja mennyi objektumot hoztunk belőle létre, és semmisítettünk meg! Ezzek a trükkel megnézhetjük, hogy elfelejtettünk-e felszabadítani listaelemet.

```
struct List
{
    int data;
    List *next;

    static int count; // !
};

int List::count = 0;

List *add(List *head, int data)
{
    if (head == 0)
    {
        List *ret = new List;
        List::count++; // !
        ret->data = data;
        ret->next = 0;
        return ret;
    }
    head->next = add(head->next, data);
    return head;
}

void free(List *head)
{
    if (head == 0)
    return;
```

```
    free(head->next);
    List::count--; // !
    delete head;
}

int main()
{
    List *head = 0;
    head = add(head, 8);
    head = add(head, 7);
    head = add(head, 2);

    free(head);
    std::cout << List::count; // !
}
```

Osztályszintű változókat csak osztályon kívül tudunk definiálni (ezek alól kivételt képeznek az osztályszintű konstans változók), ezért látható az osztály után a következő sor:

```
int List::count = 0;
```

Ezzel a kis módosítással meg is kapjuk a kívánt kimenetet: `0`. Ez alapján tudhatjuk, hogy minden objektum törlésre került.

2.2.1. Megjegyzés. A fenti módosítások csak gyakorlás célját képezték, az elkészítedő listának nem része a számláló.

Ha azonban egy elemet kétszer töröltünk, egyet meg elszivárogtattunk, az nem feltétlen nem derül ki. Ilyenkos a sanitizerek segíthetnek:

```
g++ list.cpp -fsanitize=address -g
```

A sanitizerekről bővebben lásd a 3. gyakorlat anyagát. Határozott előrelépést értünk el, de van még hova fejleszteni a listánkat. Szerencsére nem csak adattagokat, de tagfüggvényeket is tudunk struct-okba írni.

2.3. Konstruktorok

Kényelmesebbé tehetjük az életünket, ha írunk egy tagfüggvényt, mellyel kényelmesebben tudjuk létrehozni a listaelemeket:

```
struct List
{
    //tagfüggvények
    List(int _data, List *_next = NULL)
    {
        data = _data;
        next = _next;
    }

    //adattagok
    int data;
    List *next;
};
```

A fenti tagfüggvényt, vagy metódust **konstruktor**nak (*constructor*, vagy röviden *ctor*) hívjuk. A konstruktorok hozzák létre az objektumokat; vannak paraméterei, és nincs visszatérési értéke. A fenti konrtruktor még egy alapértelmezett paraméterrel is rendelkezik - ha mi csak egy `int` paraméterrel hívjuk meg a konstruktort, akkor a `_next`-et alapértelmezetten nullpointernek veszi. Mint minden tagfüggvény, ez a konstruktor is hozzáfér az adott struktúra adattagjaihoz.

Azonban a struktúránk működött eddig is, pedig nem írtunk konstruktort. Ha konstruktorra szükség van objektum létrehozáshoz, akkor hogyan lehet ez? Úgy, hogy a fordító a hiányzó kulcsfontosságú függvényeket legenerálja nekünk. Létrehoz (többek között) egy un. **default konstruktort**, ha mi explicit

nem hoztunk létre konstruktort. A default konstruktor 0 paraméterrel meghívható. Fontos azonban, ha mi írunk egy konstruktort, akkor a fordító már nem fog generálni ilyet.

Így a következőféleképpen tudunk egy `List` típusú objektumot létrehozni:

```
List head(5); //ok, létrehoz egy 5 értékkel rendelkező, 1 elemű listát
List head2; //nem ok, már nincs paraméter nélküli konstruktor
```

Egy trükkel megoldható, hogy tömörebb szintaxissal tudjuk inicializálni a listaelemeinket.

```
List(int _data, List *_next = 0) : data(_data), next(_next) {}
```

A konstruktor fejléce után (kettősponttól kezdve) található egy un. **inicializációs lista**. Az inicializációs listával rendelkező konstruktor hasonló jelentéssel bír, mint a korábbi kód, azonban inicializációs lista használata hatékonyabb.

Mire a konstruktor törzséhez ér a vezérlés, addigra az adattagokat inicializálni kell. A törzsben ezért már inicializált értékeket írunk felül, ami erőforrás pazarlás. Primitív típusok esetén ez nem jelent problémát (mivel a fordító várhatóan kioptimalizálja), összetett típusok esetén viszont számottevő lehet.

Sőt, mivel a referenciákat és konstansokat inicializálni **kell**, ezért ilyen adattagjaink csak akkor lehetnek, ha minden konstruktor inicializálja őket az inicializációs listájukban.

2.3.1. Megjegyzés. A konstruktor törzsében történő értékadás két lépés (mivel előtte egy alapértelmezett konstruktor már inicializálta az objektumot), az inicializálás csak egy.

Fontos megjegyzés, hogy az a struktúra elemei a mezők definiálásának a sorrendjében inicializálódnak. Tehát, bármilyen sorrendben írjuk mi az inicializációs listát, mindig először a `data`, és utána a `next` kerül inicializálásra. Ennek az az oka, hogy konstruktorból több is lehet, így nem lenne egyértelmű az inicializációs sorrend, függne attól, hogy mely konstruktort hívtuk meg. A mezők sorrendje ezzel szemben egyértelmű.

Ennek többek között ilyen következményei lehetnek:

```
struct Printer
{
    Printer(int i) : x(i), y(x)
    {
        std::cout << y << " " << x << std::endl; // nem definiált viselkedés
    }
    int y, x;
};

int main() { Printer a(5); }
```

Ekkor `x`-nek az értéke `y` inicializálásakor még nem definiált, így (lévén a sorrend miatt `y`-nak előbb kell értéket adni) `y` értéke is nem definiált lesz.

Előfordulhat olyan, hogy szeretnénk létrehozni egy listát, de azt szeretnénk, hogy élettartama mennél kisebb legyen. Ezt megtehetjük például úgy, ha egy külön blokkban hozzuk létre.

```
void printFirstElement(const List &l) { std::cout << l.data << std::endl; }

int main
{
    //...
    {
        List l(4);
        printFirstElement(l);
    } // l megsemmisül
    //...
}
```

Ha csak egy kifejezésben van szükségünk erre a listára, létrehozhatunk egy **név nélküli temporális változót**. Ennek a létrehozásához elhagyjuk a változó nevét, és a típus után egyből a konstruktor paramétereit adjuk meg.

```
List(4); //amint létrejön ez a változó, meg is fog semmisülni.
//...
printFirstElement(List(4)); //a lista élettartama a függvényhívástól a függ-vény futás
    ának végéig tart.
```

2.3.2. Megjegyzés. A literálok is név nélküli temporális értékek (bár nem változók). Ha pl. az `f` függvény egy darab `int`-et vár paraméterül, akkor `f(5)` hívásakor `5` egy név nélküli temporális érték.

2.3.3. Megjegyzés. A temporális változók jobb értékek, így csak konstans referenciával, vagy érték szerint tudjuk őket átvenni. Ha konstans referenciához kötjük, akkor a változó élettartama kiterjesztésre kerül, meg fog egyezni a referencia élettartamával. Bár ezt a referenciát tudjuk címképezni (azaz egy balérték lesz), ne feledjük, hogy ez a referencia csak hivatkozik egy jobbértékre, nem maga lesz az.

2.4. Destrukorok

Ahogy gondoskodtunk a listaelemek létrehozásáról, gondoskodhatnánk annak megfelelő megsemmisüléséről is.

```
struct List
{
    List(int _data, List *_next = 0) : data(_data), next(_next) {}
    ~List() // dtor
    {
        delete next;
    }

    int data;
    List *next;
};
```

Az fenti tagfüggvényt, melynél a hullámvonalat közvetlenül a struktúra neve követi **destruktor**nak (*destruktor*, röviden *dtor*) nevezzük. A destruktor mindig az objektum élettartamának végén hívódik meg, és gondoskodik a megfelelő erőforrások felszabadításáról.

A destruktort is rekurzívan írtuk meg: a **next** által mutatott memóriaterület felszabadításakor meghívja `List` típusú elem destruktorát. A lista végén a `next` egy nullpointer, azon a delete hívás nem csinál semmit.

Teszteljünk!

```
int main()
{
    List head(8);
    add(&head, 7);
    add(&head, 2);
}
```

Most úgy alakítottuk át a kódot, hogy amikor létrehozzuk a listát, akkor a fejelemet a stacken hozzuk létre, melynek értéke 8, és a pointer része nullpointer. Később az `add` függvénnyel létrehozunk a heapen egy olyan listaelemet, mely 7-et tárol, és pointer része nullpointer, és az eredeti lista fejét ráállítjuk erre.

Sikeresen elértük, hogy a lista első eleme a stack-en, de minden más eleme a heap-en legyen. Mivel olyan struktrát írtunk, mely gondoskodik arról, hogy minden dinamikusan lefoglalt területet felszabadítson, mindent csak egyszer töröl, jó sorrendben, egy *RAII* (*Resource acquisition is initialization*) osztályt írtunk. Ez Bjarne-nek egy elég szerencsétlenül választott acronymje. A lényege, hogy az adott osztály a megfelelő erőforrásokat lefoglalja magának, majd a destruktor gondoskodik az erőforrások felszabadításáról. Minden erőforrást egy stack-en lévő objektumhoz kötünk, mivel azok garantáltan automatikusan fel fognak szabadulni, a destruktoruk le fog futni. Jelen esetben a lista fejeleme, ami a stack-en van, felelős azért, hogy a heap-re allokált listaelemek felszabaduljanak a program futásának a végeztével. Így a felhasználónak már a free hívásra sem kell figyelnie.

Bjarne híres mondása, hogy a C++ szemétgyüjtéssel rendelkező nyelv, mert nem generál szemetet. A jól megírt objektumok mindig eltakarítanak maguk után.

A konstruktor/destruktor használata ugyanolyan hatékony, mintha kézzel kezeltük volna a memóriát.

Csináljunk az `add` függvényből tagfüggvényt!

```
struct List
{
    void add(int data) //eltűnt egy paraméter!
    {
        if (next == 0)
        {
            next = new List(data);
        }
        else
        {
            next->add(data);
        }
    }
    //...
};

int main()
{
    List head(8);
    head.add(7);
    head.add(2);
}
```

A nyelv egyik szépsége, hogy a felhasználónak nem kell tudnia, hogy hogyan reprezentáltuk a listát. A listát az a felhasználó is tudja használni, aki nem ismeri a heap-et, nem hallott még soha láncolt adatszerkezetekről. A későbbiekben a lista prerezentációja kicserélhető akár egy vektor szerű adatszerkezetre anélkül, hogy a felhasználói kódot módosítani kellene.

2.5. Másoló konstruktor

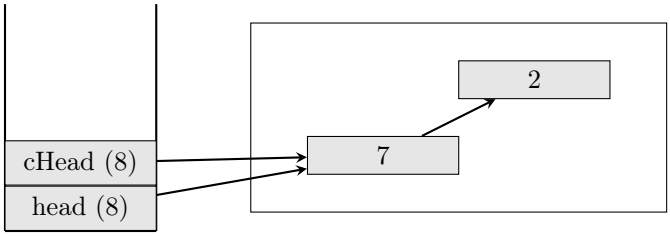
A fordító sok kódot generál a structunkba: konstruktoron és destruktoron kívül még **másoló konstruktor**t (*copy constructor*) is. A másoló konstruktor egy olyan konstrukor, melynek egyetlen paramétere egy azonos típusú objektum. Ez alapértelmezetten minden adattagot lemásol az adott adattag másoló konstruktora segítségével. Primitív típusoknál ez bitről bitre másolást jelent. Mi ennek a következménye?

```
int main()
{
    List head(8);
    head.add(7);
    head.add(2);
    {
        List cHead = head; //másoló konstruktor hívása
    } //itt lefut cHead destruktora
}
```

Fentebb létrehoztunk egy új listát `head` mintájára. A másolatnak a destruktora hamarabb lefut. Ha sanitizerrel fordítunk, futáskor hibaüzenetet kapunk: felszabadított memóriaterületet szeretnénk használni. Ennek az az oka, hogy a `cHead`-ben lévő pointer **ugyanarra** a listára fog mutatni (lévén a bitről bitre történő másolás történt a pointernél). A `cHead` megsemmisülése után a `head` destruktora megpróbál beleolvasni a már felszabadított memóriaterületbe.

A megoldás egy saját másoló konstruktor bevezetése!

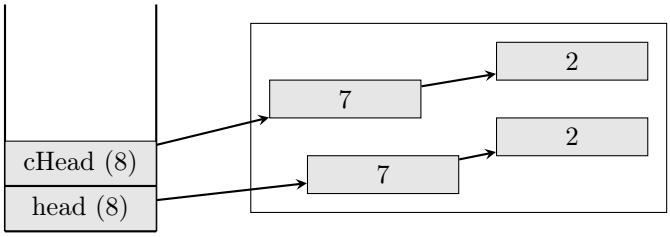
```
struct List
{
    List(const List &other) : data(other.data), next(0)
    {
```



4. ábra. A lista másolása default másoló konstruktorral. Zárójelben a lista első elemének `data` adattagjának értéke.

```
        if (other.next != 0)
        {
            next = new List(*other.next);
        }
    }
    //...
};

int main()
{
    List head(8);
    head.add(7);
    head.add(2);
    {
        List cHead = head;
    }
}
```

Mint a korábbi függvényeink, ez is rekurzív: a `new List(*other.next)` újra meghívja a copy konstruktort, ha az other.next nem nullpointer.



5. ábra. A lista másolása az általunk implementált másoló konstruktorral.

Ezzel meg is oldottuk a problémát.

Figyelem, ez egy **copy konstruktor, nem értékadás operátor!** Itt a `cHead` még nincs létrehozva, amikor head-el **inicializáljuk**. Ha az egyenlőségjel bal oldalán lévő objektum még nem jött létre, mint itt, akor a copy konstruktor hívás történik. Ellenkező esetben értékadás operátor.

```
List cHead = head; //copy ctor

List cHead;
cHead = head; //értekadás
```