C++

1. HÉT – C++1X FEATURE-ÖK

A C++ mögött máig aktív szabványosító testület áll – minden 3-4 évben új szabvány jelenik meg.

Szerencsére a 'gyártók' (akik a fordítókat készítik) a szabványt komolyan is veszik.

AZ ELSŐ NÉHÁNY HÉTBEN A C++11 ÉS AZÓTA MEGJELENT ÚJÍTÁSOKRÓL ESIK SZÓ

A C++ memória modellje (kis ismétlés)

A lokális scope-ban létrehozott változók a stack-re (verem) kerülnek

- Ha egy függvény visszatér, az általa lefoglalt lokális változók eltűnnek a stack “tetejéről”
- Ha egy függvény meghívódik, a parameter-listában kapott értékekhez létrehoz egy-egy újabb stack változót, ezekbe bemásolja az értékeket és ezek a változók lokális érvényűek

```cpp
#include <iostream>

void someFunction(int a, char b) {
  a = a+1;
  std::cout << "a + b + 1 = " << a + b << std::endl;
}

int main() {
    int a = 5;
    int b = 4;
    someFunction(a, b);
    std::cout << "a in main() = " << a << std::endl;
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
a + b + 1 = 10
a in main() = 5
```

Itt a someFunction()-ban levő a és b változók nem ugyanazok, mint a main() fv a és b változói

Ezek lokális változók amik inicializálásukkor megkapják a main() fv-ben levő a és b változók **értékét**

A C++ memória modellje (kis ismétlés)

Fentiek akkor is igazak, ha someFunction()-nak egy pointer vagy referenciát adunk át. A someFunction()-ban levő pointer vagy referencia egy új változó a stack-en – más kérdés, hogy ezek egy memória címet (pointer esetében), vagy egy másik változóra való hivatkozást tárolnak értékként

```cpp
#include <iostream>

void someFunction(int* a, int& b) {
  *a = *a+1;
  std::cout << "a + b + 1 = " << *a + b << std::endl;
  std::cout <<"a in someFunction() = " << *a << std::endl;
}

int main() {
    int a = 5;
    int b = 4;
    someFunction(&a, b);
    std::cout << "a in main() = " << a << std::endl;
    return 0;
}
```

```
› clang++-7 -pthread -std=c++17 -o main main.cpp
› ./main
a + b + 1 = 10
a in someFunction() = 6
a in main() = 6
›
```

Ez a példa tehát csak annyiban más, hogy a someFunction()-ban levő 4. sor megváltoztatja a main() fv-ben levő a értékét is.

someFunction()-ban levő a egy másik változó, de ez a külső a címét tárolja.

A C++ memória modellje (kis ismétlés)

Mi történik, ha azt szeretnénk, hogy mondjuk someFunction() létrehoz egy változót, de annak az értékét someFunction() visszatérése után is olvasni szeretnénk?

```cpp
#include <iostream>

int* someFunction(int a, int b) {
  a = a+1;
  int* sum = new int;
  *sum = a + b;
  std::cout << "a + b + 1 = " << *sum << std::endl;
  std::cout <<"a in someFunction() = " << a << std::endl;
  return sum;
}

int main() {
    int a = 5;
    int b = 4;
    int* c = someFunction(a, b);
    std::cout << "a in main() = " << a << std::endl;
    std::cout << "c in main() = " << *c << std::endl;
    delete c; // ez fontos!
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
a + b + 1 = 10
a in someFunction() = 6
a in main() = 5
c in main() = 10
```

Erre való a dinamikus memória. A dinamikusan (new operátorral) lefoglalt memória nem a stack-en, hanem a heap-en (dinamikus tárhely, free store) fog helyet kapni.

A C++ memória modellje (kis ismétlés)

Az így lefoglalt memória életciklusát a függvények meghívása / visszatérése nem befolyásolja, csak a programozó szabadíthatja fel a delete operator segítségével!

```cpp
#include <iostream>

int* someFunction(int a, int b) {
  a = a+1;
  int* sum = new int;
  *sum = a + b;
  std::cout << "a + b + 1 = " << *sum << std::endl;
  std::cout <<"a in someFunction() = " << a << std::endl;
  return sum;
}

int main() {
    int a = 5;
    int b = 4;
    int* c = someFunction(a, b);
    std::cout << "a in main() = " << a << std::endl;
    std::cout << "c in main() = " << *c << std::endl;
    delete c; // ez fontos!
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
a + b + 1 = 10
a in someFunction() = 6
a in main() = 5
c in main() = 10
```

Jóllehet, sum és c itt is lokális változók – a stack-en!

De mivel mindkettő egy pointer (egy mem.cím), az is kérdés, hogy az a cím hol van a memóriában és amit tárol, meddig él.

Erre a címre vonatkoznak a fentiek, hogy a heap-ben van és nem szűnik meg a delete meghívásáig. Ezt jól jegyezzék meg!

A C++ memória modellje (kis ismétlés)

Mindez azt is jelenti, hogy ha someFunction() nem adná vissza a lefoglalt mem.terület címét, bajban lennénk! – hiszen már nem tudnánk elérni a címet (nincsen rá handle-ünk / “fogantyúnk) és a delete-et se hívja már meg semmi.

```cpp
#include <iostream>

void someFunction(int a, int b) {
  a = a+1;
  int* sum = new int; // dangling pointer
  *sum = a + b;
  std::cout << "a + b + 1 = " << *sum << std::endl;
  std::cout <<"a in someFunction() = " << a << std::endl;
}

int main() {
    int a = 5;
    int b = 4;
    someFunction(a, b); // ezutan main()-ben mar nincs mit
    // delete-elni...
    std::cout << "a in main() = " << a << std::endl;
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
a + b + 1 = 10
a in someFunction() = 6
a in main() = 5
```

Az ilyet hívják “dangling pointer”-nek (vagy “dangling reference”-nek referencia esetén)

(A “dangling reference” kicsit máshogy jönne elő – mondjuk someFunction()-ban létrehozunk egy változót, és arra adunk vissza egy referenciát… de amikor visszatér a fv, az a változó már nem él, mivel a stackről eltűnt)

A C++ memória modellje (kis ismétlés)

Dangling reference – ilyet sajnos OOP vizsgán is sokat láttam ☹

Úgy tűnik, mintha someFunction() egy int-et adna vissza, de valójában egy int& (int referenciát) ad vissza! Ez viszont olyan dologra hivatkozik, ami a someFunction() visszatérésével megszűnik:

```cpp
#include <iostream>

int& someFunction(int a, int b) {
  int c = a + b + 1;
  return c;
}

int main() {
    int a = 5;
    int b = 4;
    int c = someFunction(a, b);
    std::cout << "a+b+1 in main() = " << c << std::endl;
    return 0;
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
main.cpp:5:10: warning: reference to stack memory associated with
      local variable 'c' returned [-Wreturn-stack-address]
  return c;
         ^
1 warning generated.
./main
a+b+1 in main() = 10
```

A C++ memória modellje (kis ismétlés)

Tehát : mindig legyünk tudatában annak, hogy mi az ami a stack-en van és mi az ami a heap-en!

Amit egy függvényben nem new-val hozunk létre, az mindig a stack-en van.

```cpp
#include <iostream>

class X {
  int a;
  int* b;
public:
  X(int aval, int bval) : a(aval), b(new int(bval)) {}
  ~X() { delete b; }
  void printSum() {std::cout << a + *b << std::endl;}
};

int main() {
    int a = 5;
    int b = 4;
    X peldany(a,b);
    peldany.printSum();
    // peldany valtozo a stack-en van
    // peldany::a a stack-en van
    // peldany::b a stack-en van
    // *peldany::b a heap-en van!
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
9
```

Más kérdés, hogy ha egy összetett objektumról van szó, akkor annak lehetnek részei amik már a heap-en kerülnek allokálásra

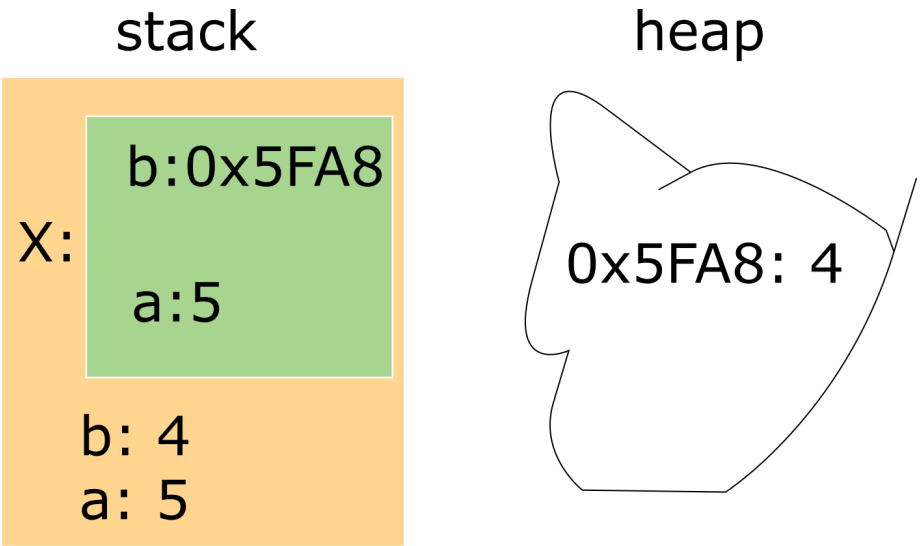
Itt például az X osztálynak van egy int* tagváltozója. Ez egy memória címet tároló változó, ami X példányosításakor a stack-en jön létre. Viszont az a memória terület, amire ez az int* hivatkozik, az már a heap-en van!

A C++ memória modellje (kis ismétlés)

A helyzet valahogy így néz ki:

```cpp
#include <iostream>

class X {
  int a;
  int* b;
public:
  X(int aval, int bval) : a(aval), b(new int(bval)) {}
  ~X() { delete b; }
  void printSum() {std::cout << a + *b << std::endl;}
};

int main() {
    int a = 5;
    int b = 4;
    X peldany(a,b);
    peldany.printSum();
    // peldany valtozo a stack-en van
    // peldany::a a stack-en van
    // peldany::b a stack-en van
    // *peldany::b a heap-en van!
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
9
```

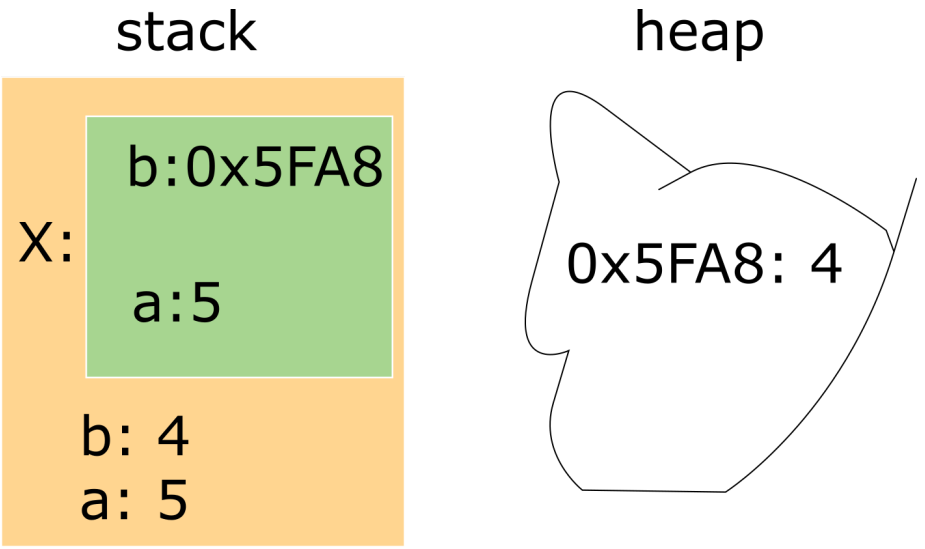


A C++ memória modellje (kis ismétlés)

Dangling pointer olyankor tud létrejönni, ha a stack-en egy változó megszűnik, de a hozzá tartozó heap memória nincs felszabadítva. Ezért fontos a destructor, amit meg is írtunk: ha egy X típusú változó a stack-en felszabadul, automatikusan meghívódik a destruktora és felszabadul a heap adott memóriaterülete is.

```cpp
#include <iostream>

class X {
  int a;
  int* b;
public:
  X(int aval, int bval) : a(aval), b(new int(bval)) {}
  ~X() { delete b; }
  void printSum() {std::cout << a + *b << std::endl;}
};

int main() {
    int a = 5;
    int b = 4;
    X peldany(a,b);
    peldany.printSum();
    // peldany valtozo a stack-en van
    // peldany::a a stack-en van
    // peldany::b a stack-en van
    // *peldany::b a heap-en van!
}
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
9
```



C++ 11 – move szemantika

Miért fontos mindez?

A C++11-es szabvány bevezette az ún. **Move semantics-et** (mozgatás szemantika)

Ez lehetővé teszi, hogy amikor egy objektumból másolatot hozunk létre, akkor klasszikus másolás helyett a dinamikusan lefoglalt memóriát ne másoljuk, hanem “ellopjuk” az újonnan létrejött másolat részére.

Ez azért hasznos, mert különösen a dinamikus memóriaterületet azért szoktuk használni, hogy előre nem ismert számú vagy méretű dolgokat tárolunk benne – és ez sok adat is lehet!

Gondoljunk egy láncolt listára, vagy egy vektorra!

- Előre nem lehet tudni, hogy hány eleme lesz.
- Lehet nagyon-nagyon sok eleme is.
- Ha egy ilyen konténert klasszikus módon másolnánk, potenciálisan nagyon sok elemet le kellene másolni
- Van, hogy ezt nem tudjuk elkerülni, mert tényleg egy másolatra van szükség. Minden más esetre ott van a move lehetősége, ami akkor használható, ha tudjuk, hogy az eredeti példányra már nem lesz szükségünk!

C++ 11 – move szemantika

Vegyünk erre egy példát. Nézzük meg, mi történik másoláskor (ez egy hosszabb példa, nézzük meg részletesen).

Először is kell egy IntNode osztály. Ennek nincs custom destruktora, mert ő nem “birtokolja” a benne levő pointert… majd az IntNode-okat menedzseli a LinkedList osztály!

```
class IntNode {
    int value;
    IntNode* next;
public:
    IntNode(int v) : value(v), next(nullptr) {}
    int getValue() { return value; }
    IntNode* getNext() { return next; }
    void setNext(IntNode* n) { next = n; }
};
```

C++ 11 – move szemantika

A LinkedList osztály IntNode-okat hoz létre, írat ki, töröl... és majd másol / mozgat is!

```
class IntLinkedList {
    IntNode* head;
public:
    IntLinkedList() : head(nullptr) {}
    ~IntLinkedList();
    void addNode(int value);
    void print();
};
```

```
IntLinkedList::~IntLinkedList() {
    IntNode* cur = head;
    IntNode* next = nullptr;
    while (true) {
        if (cur) {
            next = cur->getNext();
            delete cur;
            cur = next;
        }
        else {
            break;
        }

    }
}
```

C++ 11 – move szemantika

A LinkedList osztály IntNode-okat hoz létre, írat ki, töröl... és majd másol / mozgat is!

```
void IntLinkedList::addNode(int value) {
    if (!head) { head = new IntNode(value); }
    else {
        IntNode* pre = head;
        IntNode* cur = head->getNext();
        while (true) {
            if (!cur) {
                cur = new IntNode(value);
                if (pre) pre->setNext(cur);
                break;
            }
            pre = cur;
            cur = cur->getNext();
        }
    }
}
```

C++ 11 – move szemantika

A LinkedList osztály IntNode-okat hoz létre, írat ki, töröl... és majd másol / mozgat is!

```
void IntLinkedList::print() {
    IntNode* cur = head;
    while (true) {
        if (cur) {
            std::cout << cur->getValue();
        }
        else {
            std::cout << std::endl;
            break;
        }
        cur = cur->getNext();
        if (cur) { std::cout << ", "; }
    }
}
```

```
int main()
{
    // ll1 a stack-en van, de sok memoriara
hivatkozik a heap-en
    IntLinkedList ll1;
    ll1.addNode(5);
    ll1.addNode(6);
    ll1.addNode(7);
    ll1.print();
}
```

C++ 11 – move szemantika

Na és mi történik, ha egy ilyen objektumot másolunk? Emlékezzünk vissza, hogy a C++ osztályokhoz alapból készül copy constructor (ha mi nem készítünk) – de a rule of 3 miatt ez itt nem ideális eredményhez vezet:

```
int main()
{
    // ll1 a stack-en van, de sok memoriara hivatkozik a heap-en
    IntLinkedList ll1;
    ll1.addNode(5);
    ll1.addNode(6);
    ll1.addNode(7);
    ll1.print();
    IntLinkedList ll2(ll1); // masolat beepitett copy constr-ral
    ll2.addNode(55);
    ll1.print(); // hopp!
}
```

Ez ha nem crash-el, az utolsó sorban printelt ll1 is tartalmazni fogja az 55-öt… mivel a másolatban cím szerint ugyanaz a head pointer lesz, és innentől a teljes lista a dinamikus memóriában ugyanarra a memóriaterületre mutat.

Egyébként VS-ben crash-el is, mert ezekre a címekre most 2x hív delete-et a destruktor!

C++ 11 – move szemantika

A probléma megoldása a copy constructor (és persze assignment, ha a meglévő listák értékének felülírását is támogatni szeretnénk meglévő listák alapján) – így már jó lesz a példa is (gyakorlásként készítsék el a copy assignmentet is, hogy a rule of 3 teljesüljön!):

```
class IntLinkedList {
    IntNode* head;
public:
    IntLinkedList() : head(nullptr) {}
    ~IntLinkedList();
    IntLinkedList(const IntLinkedList& other);
    void addNode(int value);
    void print();
};
```

```
IntLinkedList::IntLinkedList(const IntLinkedList& other) : head(nullptr) {
    IntNode* curInOther = other.head;
    IntNode* preInThis = nullptr;
    while (true) {
        if (curInOther && !head) {
            head = new IntNode(curInOther->getValue());
            preInThis = head;
        }
        else if (curInOther) {
            preInThis->setNext(new IntNode(curInOther->getValue()));
            preInThis = preInThis->getNext();
        }
        else { break; }
        curInOther = curInOther->getNext();
    }
}
```

C++ 11 – move szemantika

Copy assignmentre egy lehetséges megoldás:

```
IntLinkedList& IntLinkedList::operator=(const
IntLinkedList& other) {
    if (this == &other) return *this;

    // destruktor kodja...
    IntNode* cur = head;
    IntNode* next = nullptr;
    while (true) {
        if (cur) {
            next = cur->getNext();
            delete cur;
            cur = next;
        }
        else {
            break;
        }
    }
    // idaig... es most ujra felepitjuk:
    head = nullptr;
    IntNode* curInOther = other.head;
    IntNode* preInThis = nullptr;
    while (true) {
        if (curInOther && !head) {
            head = new IntNode(curInOther->getValue());
            preInThis = head;
        }
        else if (curInOther) {
            preInThis->setNext(new IntNode(curInOther->getValue()));
            preInThis = preInThis->getNext();
        }
        else {
            break;
        }
        curInOther = curInOther->getNext();
    }
    return *this;
}
```

C++ 11 – move szemantika

Ennél szebb, ha készítünk egy külön clear() metódust, amit a destruktor is meghív...

```
void IntLinkedList::clear() {
    IntNode* cur = head;
    IntNode* next = nullptr;
    while (true) {
        if (cur) {
            next = cur->getNext();
            delete cur;
            cur = next;
        }
        else {
            break;
        }
    }
}

IntLinkedList::~IntLinkedList() {
    clear();
}
```

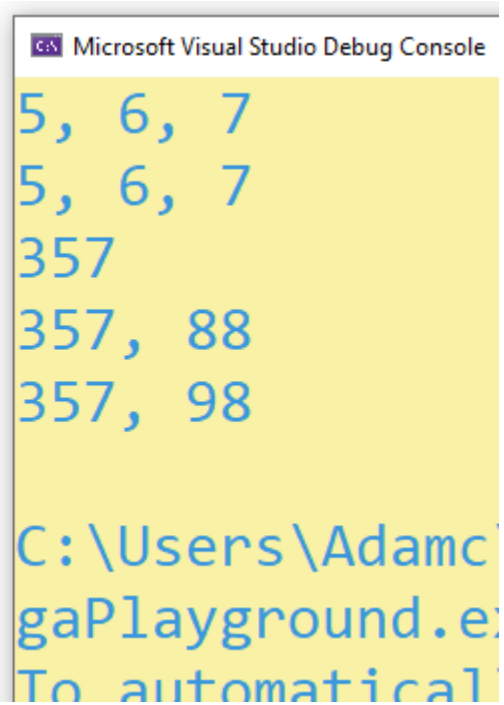
Nem mellesleg a copy-and-swap mintázat is használható:

```
IntLinkedList& IntLinkedList::operator=(const IntLinkedList& other) {
    std::cout << "\tcopy assignment called" << std::endl;
    // copy and swap
    IntLinkedList tmp(other);

    // swap resz:
    IntNode* tmpHead = tmp.head;
    tmp.head = this->head;
    this->head = tmpHead;

    return *this;
}
```

C++ 11 – move szemantika

Így most már működik a copy assignment is:

```
136  int main()
137  {
138      // ll1 a stack-en van, de sok memoriara hivatkozik a heap-en
139      IntLinkedList ll1;
140      ll1.addNode(5);
141      ll1.addNode(6);
142      ll1.addNode(7);
143      ll1.print();
144      IntLinkedList ll2(ll1); // masolat beepitett copy constr-ral
145      ll2.addNode(55);
146      ll1.print(); // mar nem hopp!
147      IntLinkedList ll3;
148      ll3.addNode(357);
149      ll3.print();
150      ll2 = ll3;
151      ll2.addNode(88);
152      ll3.addNode(98);
153      ll2.print();
154      ll3.print();
155  }
```



C++ 11 – move szemantika

A move constructor / assignment akkor lehet hasznos, ha egy X IntLinkedList alapján készíteni szeretnénk egy Y IntLinkedList-et, de mindezt úgy, hogy előre tudjuk, hogy X-re később már nem lesz szükség.

Erre tipikus példa, amikor egy függvény / metódus létrehoz egy objektumot, és annak az értékét adja vissza. A C++ ilyen ún. Return Value Optimization-t csinál (RVO), ami azt jelenti, hogy az eredményt nem lemásolja a stackről a stack egy másik változójára (ami a hívó függvény lokális változója), hanem move-olja (emlékezzünk, ez csak a dinamikus mem.részekre vonatkozik!!) RVO viszont csak akkor lehetséges, ha az osztálynak van move constructora / move assignmentje!

A move constructor és assignment szignatúrái így néznek ki (A && jelölés itt egy ún. jobb oldali referenciát jelent, amit a C++11 vezetett be):

```
IntLinkedList(IntLinkedList&& other);
IntLinkedList& operator=(IntLinkedList&& other);
```

C++ 11 – move szemantika

Tegyük fel például, hogy van egy ilyen metódusunk:

```
IntLinkedList IntLinkedList::mapAddOne() {
    IntLinkedList retval;
    IntNode* cur = head;
    while (true) {
        if (cur) {
            retval.addNode(cur->getValue() + 1);
        }
        else { break; }
        cur = cur->getNext();
    }
    return retval;
}
```

Ennek használatakor sajnos teljesen feleslegesen a copy constructor fog meghívódni, mielőtt visszatér:

```
357, 88
357, 98
 -> before mapAddOne() return
        copy constructor called
358, 99
```

C++ 11 – move szemantika

A hatékonyabb működéshez készítsünk move constructort:

```
IntLinkedList::IntLinkedList(IntLinkedList&& other) : head(other.head) {
    std::cout << "\tmove constructor called" << std::endl;
    other.head = nullptr;
}
```

Ennek megvalósítása nagyon egyszerű. Egyszerűen beállítjuk az új objektum head változójának értékét (a címet) a mozgatandó objektum head változójának értékére.

A kettő innentől ugyanarra a címre hivatkozik.

Azt viszont ne felejtsük el, hogy a függvény meghívása után other megszűnhet (meghívódhat a destruktora!) Pl. ebben a példában retval meg fog szűnni (a mapAddOne() metódus ha visszatér)! Ezért beállítjuk other.head értékét nullptr-re, így ha delete hívódik rá, nem történik semmi.

C++ 11 – move szemantika

A move assignment is nagyon hasonló (hogy melyik hívódik, attól függ, hogy a cél objektum most jön-e létre vagy már létezik). Viszont ilyenkor a már meglévő tartalmat is törölni kell, hogy biztosan ne legyen memóriaszivárgás:

```
IntLinkedList& IntLinkedList::operator=(IntLinkedList&& other) {
    std::cout << "\tmove assignment called" << std::endl;
    // destruktor kodja...
    IntNode* cur = head;
    IntNode* next = nullptr;
    while (true) {
        if (cur) {
            next = cur->getNext();
            delete cur;
            cur = next;
        }
        else {
            break;
        }
    }
    // innentol meg csak atirjuk a pointer cimet:
    head = other.head;
    other.head = nullptr;
    return *this;
}
```

C++ 11 – move szemantika

Megjegyezzük, hogy a korszerű fordítók manapság már sokszor move constructor / assignment nélkül is képesek RVO-re…

- egyszerűen megoldják, hogy a visszatérő változó értéke már eleve a hívó függvény scope-jában fenntartott változóba íródjon

Mindez persze nem jelenti azt, hogy haszontalan lenne a move constructor / assignment, hiszen lehetnek olyan esetek, ahol a fordító nem tudja kideríteni, hol jön létre az adott érték, illetve majd látni fogjuk, hogy lesznek esetek, ahol mi magunk szeretnénk majd garantálni, hogy valahol másolás helyett mozgatás történjen

C++ 11 – move szemantika

Végezetül: fontos, hogy a move constructor és move assignment garantáltan ne dobjon kivételt! Gondoljunk bele, mi történne, ha mozgatás közben félúton elhasalna a művelet, és egy catch blokkban kellene folytatnia a programnak a futást.

Ilyenkor már az eredeti objektum se lenne érintetlen állapotban. Ezért, annak érdekében hogy a fordító automatikusan használja a move-ot, sokszor kikötés, hogy a noexcept módosító is szerepeljen a move constructor / move assignment fejlécében:

```
IntLinkedList& IntLinkedList::operator=(IntLinkedList&& other) noexcept {
    std::cout << "\tmove assignment called" << std::endl;
    // destruktor kodja...
    IntNode* cur = head;
    IntNode* next = nullptr;
    while (true) {
        if (cur) {
            next = cur->getNext();
            delete cur;
            cur = next;
        }
        // ...
```

Referenciák típusai

Láthattuk, hogy a move constructor / move assignment esetén egy újfajta referencia jelent meg (a && jelölésű jobb oldali referencia), amivel korábban még nem találkoztunk. Vegyük most át az összes referencia típust!

A C++ nyelv 2011 óta 3-féle referenciát különböztet meg:

- Lvalue (bal oldali) referencia: hivatkozás T típusra T& jelöléssel
- Const (konstans) referencia: hivatkozás T típusra const T& jelöléssel – ilyenkor az adott érték csak olvasható. Mégis hasznos, mert – különösen ha nagy objektumról van szó – jó dolog, ha nem kell a stack-en másolgatni
- Rvalue (jobb oldali) referencia: hivatkozás T típusra T&& jelöléssel. Az rvalue referencia azt mondja a fordítónak, hogy az adott referencia egy temporary (vagy expiring) változó értékére hivatkozik, amely változóra később nem lesz szükség. Miért jó ez?
  - Mint láthattuk, a változó által birtokolt erőforrások memóriaterülete másolás nélkül átadható egy másik változónak
  - Úgy érdemes felfogni, hogy van egy memóriarekeszem, ami tárol egy értéket. Ha tudom, hogy az adott változót már nem fogja a program használni, az értéket átadhatom egy másik változónak anélkül, hogy le kellene másolnom.

Referenciák típusai

Például: vessük össze az alábbi két swap() függvényt. Itt az a különbség, hogy std::move(x) egy jobb oldali referenciát csinál x változóból

```
void swap1(std::string& a, std::string& b) {
  std::string temp{ a };
  a = b;
  b = temp;
}
```

```
void swap2(std::string& a, std::string& b) {
  std::string temp{ std::move(b) };
  b = std::move(a);
  a = std::move(temp);
}
```

Ezért, míg a bal oldalon temp változó új dinamikus memóriát foglal le, a jobb oldalon egyszerűen felhasználja b korábbi dinamikus memória területét. Az új b pedig felhasználja a korábbi dinamikus memória területét. Az új a pedig felhasználja temp (vagyis b korábbi) dinamikus memória területét.

Referenciák típusai

Foglaljuk kicsit most össze más szempontból is, mit tudunk a referenciákról!

A referenciákat mindig inicializálni kell – hiszen nem változók hanem valamilyen (bal oldali vagy temporary) értékre hivatkoznak!

Bal oldali referenciát csak bal oldali értékkel (tehát pl. egy változó nevével) inicializálhatunk

Konstans referenciát inicializálhatunk bal oldali értékkel és jobb oldali értékkel is

Jobb oldali referenciát csak jobb oldali értékkel inicializálhatunk (pl. egy konstans literállal) – de persze, mint láttuk az std::move() segítségével jelezhetjük a fordítónak, hogy egy adott változót kezelhet ellopható jobb oldali értékként is.

A referenciákban közös, hogy mindegyik egyfajta hivatkozás – a másolást megspóroljuk velük. De a jobb oldali referencia kifejezetten a mozgatást támogatja, a bal és konstans referencia pedig a költséghatékony átadását értékeknek (úgy hogy írhatjuk, vagy úgy hogy csak olvashatjuk).

Referenciák típusai

Végezetül, fontos arról szót ejteni, hogy mi történik, ha egy referencia referenciáját vesszük! (ez a későbbiek miatt lesz fontos)

Ilyenkor eredményül csak akkor kapunk jobb oldali referenciát, ha jobb oldali referencia jobb oldali referenciáját vesszük!

Minden más esetben sima, bal oldali referenciát kapunk (tehát egy bal oldali referencia bármilyen referenciája bal oldali marad, és egy jobb oldali referencia bal oldali referenciáját is bal oldali referenciaként értelmez a fordító. Ezeket a szabályokat úgy nevezik, hogy **“reference collapsing rules”, és mindezt akkor lesz fontos értenünk, amikor olyan függvényeket írunk, melyek jobb oldali referenciát várnak, de ha bal oldali referenciát adunk nekik át akkor a reference collapse miatt bal oldali referenciaként is kezelhetik a bemenő típust. (ez majd a perfect forwarding témakörnél jön elő).**

Referenciák típusai

Kétféle swap() profilozása: https://coliru.stacked-crooked.com/a/4c49d48428fd804b

```cpp
#include <string>
#include <chrono>
#include <iostream>


void swap1(std::string& a, std::string& b) {
  std::string temp{ a };
  a = b;
  b = temp;
}

void swap2(std::string& a, std::string& b) {
  std::string temp{ std::move(b) };
  b = std::move(a);
  a = std::move(temp);
}


int main() {
    auto start1 = std::chrono::high_resolution_clock::now();
    std::string s1("stringa");
    std::string s2("stringb");
    for (int i = 0; i < 100000; i++) {
        swap1(s1, s2);
    }

    auto end1 = std::chrono::high_resolution_clock::now();
    std::cout << std::chrono::duration_cast<std::chrono::microseconds>(end1 - start1).count() << std::endl;

    auto start2 = std::chrono::high_resolution_clock::now();
    std::string s3("stringc");
    std::string s4("stringd");
    for (int i = 0; i < 100000; i++) {
        swap2(s3, s4);
    }

    auto end2 = std::chrono::high_resolution_clock::now();
    std::cout << std::chrono::duration_cast<std::chrono::microseconds>(end2 - start2).count() << std::endl;


}
```

```
g++ -std=c++17 -O2 -Wall -pedantic -pthread main.cpp && ./a.out
7271
4183
```