Informatika 3.
3. előadás: Dinamikus memóriakezelés

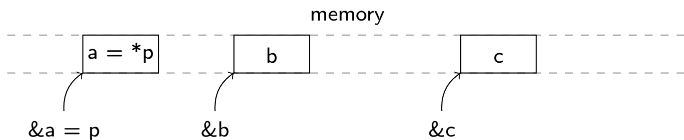
Kovács Kristóf

Budapesti Műszaki és Gazdaságtudományi Egyetem

2024-03-05

Ismétlés

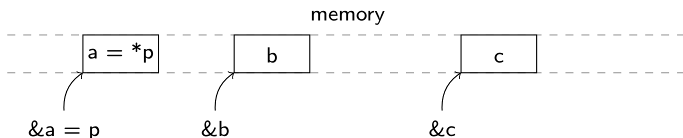
- Pointerekkel közvetlenül memória területekre hivatkozhatunk




```
int a, b, c;
int *p = &a;
*p = 5
```

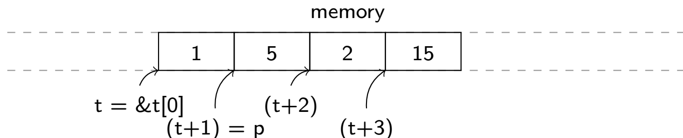
Ismétlés

- Pointerekkel közvetlenül memória területekre hivatkozhatunk




```
int a, b, c;
int *p = &a;
*p = 5
```

- A tömbök pointerek, a pointerek tömbök




```
int t[] = {1, 5, 2, 15};
int *p = t + 1;
int x = *(t + 1); // 5
```

Dinamikus memóriafoglalás

- Ettől a ponttól C++ kódot írunk.

Dinamikus memóriafoglalás

- Ettől a ponttól C++ kódot írunk.
- Tudunk manuálisan memóriát foglalni a **new** kulcsszóval, ez egy pointert ad vissza, ami a lefoglalt területre mutat:

```
int *p = new int;
*p = 5;
cout << "pointer: " << p << " mem: " << *p << endl;
// pointer: 0xa000004d0 mem: 5
```

Dinamikus memóriafoglalás

- Ettől a ponttól C++ kódot írunk.
- Tudunk manuálisan memóriát foglalni a **new** kulcsszóval, ez egy pointert ad vissza, ami a lefoglalt területre mutat:

```
int *p = new int;
*p = 5;
cout << "pointer: " << p << " mem: " << *p << endl;
// pointer: 0xa000004d0 mem: 5
```

- Itt már a C++-os kiírást használom. Az **stdio.h**-s **printf** és **scanf**-et nem fogjuk innentől kezdve használni.

Dinamikus memóriafoglalás

- Ettől a ponttól C++ kódot írunk.
- Tudunk manuálisan memóriát foglalni a **new** kulcsszóval, ez egy pointert ad vissza, ami a lefoglalt területre mutat:

```
int *p = new int;
*p = 5;
cout << "pointer: " << p << " mem: " << *p << endl;
// pointer: 0xa000004d0 mem: 5
```

- Itt már a C++-os kiírást használom. Az **stdio.h**-s **printf** és **scanf**-et nem fogjuk innentől kezdve használni.
- Dinamikus memóriafoglalás létezik C-ben is, de felesleges lenne a bonyolultabb szintaxisát megtanulnunk.

C++ fordítás

- Szinte semmiben nem változik meg a fordítás menete.

C++ fordítás

- Szinte semmiben nem változik meg a fordítás menete.
- Akik fejlesztői környezetben dolgoznak (Codeblocks, Visual Studio, stb.) **.c** helyett **.cpp** kiterjesztéssel nevezzék el a file-okat és ennyi.

C++ fordítás

- Szinte semmiben nem változik meg a fordítás menete.
- Akik fejlesztői környezetben dolgoznak (Codeblocks, Visual Studio, stb.) **.c** helyett **.cpp** kiterjesztéssel nevezzék el a file-okat és ennyi.
- Akik parancssorból dolgoznak **gcc** helyett **g++**-t használjanak.

iostream

- Tegyünk egy kis kitérőt, hogy megismerjük az új kiírást.

iostream

- Tegyünk egy kis kitérőt, hogy megismerjük az új kiírást.
- Két könnyen megjegyezhető kulcsszó: **cin** és **cout**.

iostream

- Tegyünk egy kis kitérőt, hogy megismerjük az új kiírást.
- Két könnyen megjegyezhető kulcsszó: **cin** és **cout**.
- **cin**-el tudunk beolvasni értékeket, ez helyettesíti innentől a **scanf**-et:

```
int a;
float f;
cin >> a;
cin >> f;
```

iostream

- Tegyünk egy kis kitérőt, hogy megismerjük az új kiírást.
- Két könnyen megjegyezhető kulcsszó: cin és cout.
- cin-el tudunk beolvasni értékeket, ez helyettesíti innentől a scanf-et:

```
int a;
float f;
cin >> a;
cin >> f;
```

- Nem kell jelezni, hogy milyen típust olvasunk be, így sokkal kényelmesebb mint a scanf.

iostream

- Tegyünk egy kis kitérőt, hogy megismerjük az új kiírást.
- Két könnyen megjegyezhető kulcsszó: **cin** és **cout**.
- **cin**-el tudunk beolvasni értékeket, ez helyettesíti innentől a **scanf**-et:

```
int a;
float f;
cin >> a;
cin >> f;
```

- Nem kell jelezni, hogy milyen típust olvasunk be, így sokkal kényelmesebb mint a **scanf**.
- A **cout**-hoz hasonlóan nem kell típus és lehet "fűzni":

```
int a = 5;
float f = 6.4;
cout << "a: " << a << endl << "f: " << f << endl;
```

iostream

- Tegyünk egy kis kitérőt, hogy megismerjük az új kiírást.
- Két könnyen megjegyezhető kulcsszó: **cin** és **cout**.
- **cin**-el tudunk beolvasni értékeket, ez helyettesíti innentől a **scanf**-et:

```
int a;
float f;
cin >> a;
cin >> f;
```

- Nem kell jelezni, hogy milyen típust olvasunk be, így sokkal kényelmesebb mint a **scanf**.
- A **cout**-hoz hasonlóan nem kell típus és lehet "fűzni":

```
int a = 5;
float f = 6.4;
cout << "a: " << a << endl << "f: " << f << endl;
```

- Az **endl** az új sor (end line).

iostream

- A **cin** és **cout** az **iostream** könyvtárban vannak definiálva. Mostantól ezt használjuk az **stdio.h** helyett.

iostream

- A **cin** és **cout** az **iostream** könyvtárban vannak definiálva. Mostantól ezt használjuk az **stdio.h** helyett.
- A C++-os könyvtárakat kiterjesztés nélkül használjuk, így lettek megkülönböztetve a C-s könyvtáraktól (ahol a .h kiterjesztés szokás).

iostream

- A **cin** és **cout** az **iostream** könyvtárban vannak definiálva. Mostantól ezt használjuk az **stdio.h** helyett.
- A C++-os könyvtárakat kiterjesztés nélkül használjuk, így lettek megkülönböztetve a C-s könyvtáraktól (ahol a .h kiterjesztés szokás).
- Egy teljes program **cin** és **cout**-al:

```
1  #include <iostream>
2  using namespace std;
3  int main(void) {
4    float x;
5    cin >> x;
6    cout << "x erteke: " << x << endl;
7    return 0;
8  }
```

iostream

- A **cin** és **cout** az **iostream** könyvtárban vannak definiálva. Mostantól ezt használjuk az **stdio.h** helyett.
- A C++-os könyvtárakat kiterjesztés nélkül használjuk, így lettek megkülönböztetve a C-s könyvtáraktól (ahol a .h kiterjesztés szokás).
- Egy teljes program **cin** és **cout**-al:

```
1  #include <iostream>
2  using namespace std;
3  int main(void) {
4    float x;
5    cin >> x;
6    cout << "x erteke: " << x << endl;
7    return 0;
8  }
```

- A **using namespace std** sorról később lesz szó, addig csak másoljuk be mindig az **iostream** alá.

Dinamikusan foglalt tömb

- Tömböt is foglalhatunk dinamikusan:

```
int *t;
t = new int[3];
t[0] = 5; t[1] = 6; t[2] = 7;
```

Dinamikusan foglalt tömb

- Tömböt is foglalhatunk dinamikusan:

```
int *t;
t = new int[3];
t[0] = 5; t[1] = 6; t[2] = 7;
```

- A dinamikusan foglalt tömb elemei is egymás mellé kerülnek a memóriában.

Dinamikusan foglalt tömb

- Tömböt is foglalhatunk dinamikusan:

```
int *t;
t = new int[3];
t[0] = 5; t[1] = 6; t[2] = 7;
```

- A dinamikusan foglalt tömb elemei is egymás mellé kerülnek a memóriában.
- Így már lehetséges felhasználó által megadott méretű tömböt létrehozni:

```
1 int *t;
2 int n;
3 cin >> n;
4 t = new int[n];
5 for(int i = 0; i < n; i++) {
6   t[i] = i * i;
7 }
```

Dinamikusan foglalt tömb

- Tömböt is foglalhatunk dinamikusan:

```
int *t;
t = new int[3];
t[0] = 5; t[1] = 6; t[2] = 7;
```

- A dinamikusan foglalt tömb elemei is egymás mellé kerülnek a memóriában.
- Így már lehetséges felhasználó által megadott méretű tömböt létrehozni:

```
1 int *t;
2 int n;
3 cin >> n;
4 t = new int[n];
5 for(int i = 0; i < n; i++) {
6   t[i] = i * i;
7 }
```

- Új feature: lehet a **for**-ban létrehozni a ciklus változót.

Tömbbel való visszatérés

- Próbáljunk meg egy tömbböt visszaadni egy függvénnyel

Tömbbel való visszatérés

- Próbáljunk meg egy tömböt visszaadni egy függvénnyel

```
int* fv() {
  int t[] = {1, 2, 5};
  return t;
}
int main(void) {
  int *a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  return 0;
}
```

Tömbbel való visszatérés

- Próbáljunk meg egy tömbböt visszaadni egy függvénnyel

```
int* fv() {
  int t[] = {1, 2, 5};
  return t;
}
int main(void) {
  int *a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  return 0;
}
```

- Ez `Segmentation Fault` hibához vezet.

Tömbbel való visszatérés

- Próbáljunk meg egy tömbböt visszaadni egy függvénnyel

```
int* fv() {
  int t[] = {1, 2, 5};
  return t;
}
int main(void) {
  int *a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  return 0;
}
```

- Ez `Segmentation Fault` hibához vezet.
- A **t** tömb megszűnik amikor visszatér az **fv** függvény, mint bármi más lokális változó.

Tömbbel való visszatérés

- Próbáljunk meg egy tömbböt visszaadni egy függvénnyel

```
int* fv() {
  int t[] = {1, 2, 5};
  return t;
}
int main(void) {
  int *a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  return 0;
}
```

- Ez `Segmentation Fault` hibához vezet.
- A **t** tömb megszűnik amikor visszatér az **fv** függvény, mint bármi más lokális változó.
- Amivel visszatér a függvény az a **t** pointer másolata.

Tömbbel való visszatérés 2

- Minden ami dinamikusan foglalt megmarad a scope-ján kívül is.

Tömbbel való visszatérés 2

- Minden ami dinamikusan foglalt megmarad a scope-ján kívül is.
- Ez azt jelenti, hogy a következő már működik:

```
int* fv() {
  int *t = new int[3];
  t[0] = 1; t[1] = 2; t[2] = 5;
  return t;
}
int main(void) {
  int *a;
  a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  return 0;
}
```

Tömbbel való visszatérés 2

- Minden ami dinamikusan foglalt megmarad a scope-ján kívül is.
- Ez azt jelenti, hogy a következő már működik:

```
1   int* fv() {
2     int *t = new int[3];
3     t[0] = 1; t[1] = 2; t[2] = 5;
4     return t;
5   }
6   int main(void) {
7     int *a;
8     a = fv();
9     for(int i = 0; i < 3; i++) {
10      cout << a[i] << endl;
11    }
12    return 0;
13  }
```

- Igazából a dinamikusan foglalt dolgok akár a program futása végéig megmaradnak.

Tömbbel való visszatérés 3

- Mi a probléma ezzel?

```
int* fv() {
  int *t = new int[3];
  t[0] = 1; t[1] = 2; t[2] = 5;
  int *t_sqr = new int[3];
  for(int i = 0; i < 3; i++) {
    t_sqr[i] = t[i] * t[i];
  }
  return t_sqr;
}
```

Tömbbel való visszatérés 3

- Mi a probléma ezzel?

```
int* fv() {
  int *t = new int[3];
  t[0] = 1; t[1] = 2; t[2] = 5;
  int *t_sqr = new int[3];
  for(int i = 0; i < 3; i++) {
    t_sqr[i] = t[i] * t[i];
  }
  return t_sqr;
}
```

- A dinamikusan foglalt **t** tömb a függvény lefuttatása után a memóriában marad, de semmi módunk nem lesz az elérésére.

Tömbbel való visszatérés 3

- Mi a probléma ezzel?

```
int* fv() {
  int *t = new int[3];
  t[0] = 1; t[1] = 2; t[2] = 5;
  int *t_sqr = new int[3];
  for(int i = 0; i < 3; i++) {
    t_sqr[i] = t[i] * t[i];
  }
  return t_sqr;
}
```

- A dinamikusan foglalt **t** tömb a függvény lefuttatása után a memóriában marad, de semmi módunk nem lesz az elérésére.
- Ha a függvényt lefuttatjuk 100-szor, akkor 100 különböző helyen marad bent a memóriában feleslegesen.

Memória felszabadítása

- A dinamikusan foglalt dolgokat manuálisan kell felszabadítanunk (törölnünk a memóriából).

Memória felszabadítása

- A dinamikusan foglalt dolgokat manuálisan kell felszabadítanunk (törölnünk a memóriából).
- Erre van a **delete** kulcsszó:

```
int p* = new int;
*p = 5;
delete p;
```

Memória felszabadítása

- A dinamikusan foglalt dolgokat manuálisan kell felszabadítanunk (törölnünk a memóriából).
- Erre van a **delete** kulcsszó:

```
int p* = new int;
*p = 5;
delete p;
```

- És dinamikusan foglalt tömbök felszabadítására a **delete[]** kulcsszó

```
1  int* fv() {
2    int *t = new int[3];
3    t[0] = 1; t[1] = 2; t[2] = 5;
4    int *t_sqr = new int[3];
5    for(int i = 0; i < 3; i++) {
6      t_sqr[i] = t[i] * t[i];
7    }
8    delete[] t;
9    return t_sqr;
10 }
```

Memória felszabadítása 2

- Mivel az **fv** függvényünk dinamikusan foglalt tömböt ad vissza, így azt a meghívó függvénynek a felelőssége felszabadítani (ezesetben a **main**-nek):

```
1  int main(void) {
2    int *a;
3    a = fv();
4    for(int i = 0; i < 3; i++) {
5      cout << a[i] << endl;
6    }
7    delete[] a;
8    return 0;
9  }
```

Memória felszabadítása 2

- Mivel az **fv** függvényünk dinamikusan foglalt tömböt ad vissza, így azt a meghívó függvénynek a felelőssége felszabadítani (ezesetben a **main**-nek):

```
int main(void) {
  int *a;
  a = fv();
  for(int i = 0; i < 3; i++) {
    cout << a[i] << endl;
  }
  delete[] a;
  return 0;
}
```

- Érdemes erre úgy gondolni, hogy minden dinamikus foglalásnak kell lennie felszabadítás párjának.

Dinamikus memória előnyei

- De miért is helyezünk ekkora hangsúlyt a pointerekre és dinamikusan foglalt memóriára?

Dinamikus memória előnyei

- De miért is helyezünk ekkora hangsúlyt a pointerekre és dinamikusan foglalt memóriára?
- Amire használtuk eddig:

Dinamikus memória előnyei

- De miért is helyezünk ekkora hangsúlyt a pointerekre és dinamikusan foglalt memóriára?
- Amire használtuk eddig:
  - Függvényekkel megváltoztatni külső változókat (vagy visszaadni több mint 1 értéket).

Dinamikus memória előnyei

- De miért is helyezünk ekkora hangsúlyt a pointerekre és dinamikusan foglalt memóriára?
- Amire használtuk eddig:
  - Függvényekkel megváltoztatni külső változókat (vagy visszaadni több mint 1 értéket).
  - Tömböket bonyolultabban kezelni.

Dinamikus memória előnyei

- De miért is helyezünk ekkora hangsúlyt a pointerekre és dinamikusan foglalt memóriára?
- Amire használtuk eddig:
  - Függvényekkel megváltoztatni külső változókat (vagy visszaadni több mint 1 értéket).
  - Tömböket bonyolultabban kezelni.
  - Dinamikusan foglalt memória címét átvenni.

Dinamikus memória előnyei

- De miért is helyezünk ekkora hangsúlyt a pointerekre és dinamikusan foglalt memóriára?
- Amire használtuk eddig:
  - Függvényekkel megváltoztatni külső változókat (vagy visszaadni több mint 1 értéket).
  - Tömböket bonyolultabban kezelni.
  - Dinamikusan foglalt memória címét átvenni.
  - Tömböt közvetlenül visszaadni függvénnyel.

Dinamikus memória előnyei

- De miért is helyezünk ekkora hangsúlyt a pointerekre és dinamikusan foglalt memóriára?
- Amire használtuk eddig:
  - Függvényekkel megváltoztatni külső változókat (vagy visszaadni több mint 1 értéket).
  - Tömböket bonyolultabban kezelni.
  - Dinamikusan foglalt memória címét átvenni.
  - Tömböt közvetlenül visszaadni függvénnyel.
- Most egy új adatszerkezet, egy dinamikus adatszerkezet készítésére fogjuk felhasználni.

A probléma

- Dolgozzunk ki egy adatszerkezetet, ami képes tetszőlegesen sok elemet tárolni (amíg a gépünk memóriája engedi)!

A probléma

- Dolgozzunk ki egy adatszerkezetet, ami képes tetszőlegesen sok elemet tárolni (amíg a gépünk memóriája engedi)!
- Amit szeretnénk az valami hasonló adatszerkezet, mint a lista pythonban. Működjön a következő vagy valami nagyon hasonló:

A probléma

- Dolgozzunk ki egy adatszerkezetet, ami képes tetszőlegesen sok elemet tárolni (amíg a gépünk memóriája engedi)!
- Amit szeretnénk az valami hasonló adatszerkezet, mint a lista pythonban. Működjön a következő vagy valami nagyon hasonló:

```
1  int x;          // segédváltozó
2  list l;         // az általunk kitalált adatszerkezet
3  append(l, 5) // a lista végére beillesztjük az 5-öt
4  append(l, 4) // most pedig a 4-et
5  cin >> x;
6  while(x != 0) { // amíg 0-t nem kapunk
7    append(l, x)  // addig adjuk l-hez a számokat
8    cin >> x;   // beolvassuk a következő számot
9  }
```

Tömb bővítés ötlet

Első ötlet:

- Használjunk dinamikusan foglalt tömböt.
- Tároljuk egy változóban hogy jelenleg mekkora a tömb mérete.
- Ha több hely kell, foglaljunk le nagyobb tömböt és másoljuk át az eddigi értékeket az új tömbbe, a régit szabadítsuk fel.

Tömb bővítés ötlet

Első ötlet:

- Használjunk dinamikusan foglalt tömböt.
- Tároljuk egy változóban hogy jelenleg mekkora a tömb mérete.
- Ha több hely kell, foglaljunk le nagyobb tömböt és másoljuk át az eddigi értékeket az új tömbbe, a régit szabadítsuk fel.

A problémák:

- Továbbra is limitált, hogy mekkora lehet a tömb, mert az **int** és minden más egész változó típusnak fix a mérete. Van "int max" aminél nagyobb szám nem fér bele. Így nem tudnánk indexelni a tömböt ezen túl.
- Minden bővítésnél rengeteg másolás történik, ez költséges lehet hosszútávon.
- Akár létrehozhatnánk egy "int max" méretű tömböt és ugyanitt lennénk.

Bővülő tömb megvalósítása minimálisan (csak érdekesség)

(A **x->y** ekvivalens a **(*x).y**-al, később lesz róla szó.)

```
struct list {
  int *a;
  int n;
  int max;
};

void append(struct list *l, int e) {
  if(l->n >= l->max) {
    int *t = new int[l->max + 100];
    for(int i = 0; i < l->n; i++) {
      t[i] = l->a[i];
    }
    delete[] l->a;
    l->a = t;
    l->max = l->max + 100;
  }
  l->a[l->n] = e;
  l->n++;
}
```

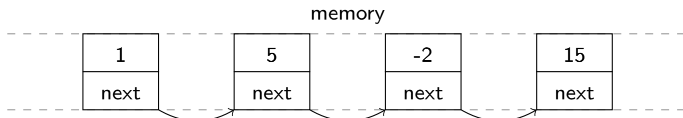
```
int main(void) {
  struct list l;
  l.a = new int[1];
  l.n = 0;
  l.max = 1;
  append(&l, 1);
  append(&l, 5);
  append(&l, -2);
  for(int i = 0; i < l.n; i++) {
    cout << l.a[i] << endl;
  }
  delete[] l.a;
  return 0;
}
```

Láncolt lista ötlet

- Mi lenne ha a lista szerkezetünk minden eleme tárolná a rákövetkező elemre mutató pointert?

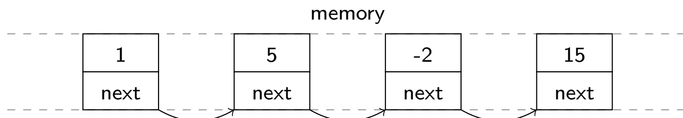
Láncolt lista ötlet

- Mi lenne ha a lista szerkezetünk minden eleme tárolná a rákövetkező elemre mutató pointert?
- Valahogy így:



Láncolt lista ötlet

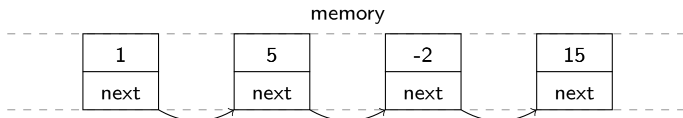
- Mi lenne ha a lista szerkezetünk minden eleme tárolná a rákövetkező elemre mutató pointert?
- Valahogy így:



- Ez az 1, 5, -2, 15 elemeket tárolná és elég lenne tudni az első elemre a pointert ahhoz, hogy bármelyiket elérjük.

Láncolt lista ötlet

- Mi lenne ha a lista szerkezetünk minden eleme tárolná a rákövetkező elemre mutató pointert?
- Valahogy így:

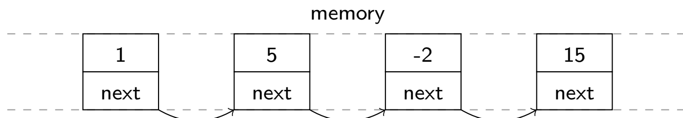


- Ez az 1, 5, -2, 15 elemeket tárolná és elég lenne tudni az első elemre a pointert ahhoz, hogy bármelyiket elérjük.
- Így nézne ki egy elem **struct**-ja:

```
struct list_e {
  int num;
  struct list_e *next;
};
```

Láncolt lista ötlet

- Mi lenne ha a lista szerkezetünk minden eleme tárolná a rákövetkező elemre mutató pointert?
- Valahogy így:



- Ez az 1, 5, -2, 15 elemeket tárolná és elég lenne tudni az első elemre a pointert ahhoz, hogy bármelyiket elérjük.
- Így nézne ki egy elem **struct**-ja:

```
struct list_e {
  int num;
  struct list_e *next;
};
```

- Tárol egy értéket a **num**-ban és a következő ugyanilyen típusú lista elem pointerét a **next**-ben.

Láncolt lista megvalósítása

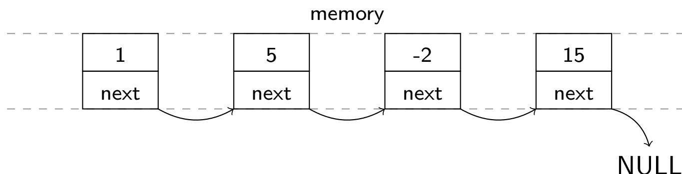
- Valósítsuk ezt meg. De rögtön jön egy probléma. Mire mutasson az utolsó elem aminek nincs rákövetkezője?

Láncolt lista megvalósítása

- Valósítsuk ezt meg. De rögtön jön egy probléma. Mire mutasson az utolsó elem aminek nincs rákövetkezője?
- Használjuk a C stringek lezáró **'\0'** karakter ötletét.

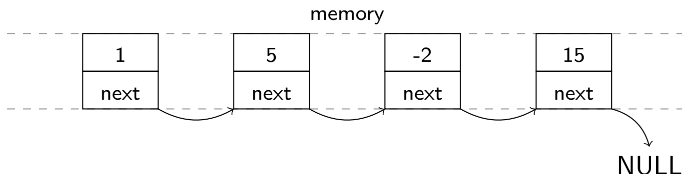
Láncolt lista megvalósítása

- Valósítsuk ezt meg. De rögtön jön egy probléma. Mire mutasson az utolsó elem aminek nincs rákövetkezője?
- Használjuk a C stringek lezáró **'\0'** karakter ötletét.
- A **NULL** pointer egy speciális pointer érték, amivel jelölhetjük, ha valami semmire nem mutat:

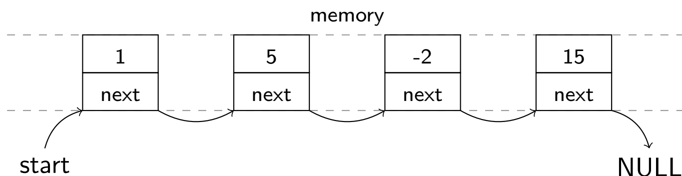


Láncolt lista megvalósítása

- Valósítsuk ezt meg. De rögtön jön egy probléma. Mire mutasson az utolsó elem aminek nincs rákövetkezője?
- Használjuk a C stringek lezáró **'\0'** karakter ötletét.
- A **NULL** pointer egy speciális pointer érték, amivel jelölhetjük, ha valami semmire nem mutat:



- Valamint egy változóban tároljuk az első elem pointerét és elméletben készen is vagyunk:



Láncolt lista megvalósítása

- Írjuk meg ehhez a C++ kódot. Azt szeretnénk, hogy ez a main függvény működjön:

```
1  int main(void) {
2    struct list_e *start = NULL; // pointer az első elemre
3    append(start, 1);            // hozzáadjuk az első elemet
4    append(start, 5);            // majd a másodikat
5    append(start, -2);           // és még egy harmadikat
6    for(;;) {    // valahogy végig kell majd menni a listán
7      // itt irassuk ki a lista elemeit
8    }
9    return 0;
10 }
```

Láncolt lista megvalósítása

- Írjuk meg ehhez a C++ kódot. Azt szeretnénk, hogy ez a main függvény működjön:

```
1  int main(void) {
2    struct list_e *start = NULL; // pointer az első elemre
3    append(start, 1);            // hozzáadjuk az első elemet
4    append(start, 5);           // majd a másodikat
5    append(start, -2);         // és még egy harmadikat
6    for(;;) {    // valahogy végig kell majd menni a listán
7      // itt irassuk ki a lista elemeit
8    }
9    return 0;
10 }
```

- Kezdjük el megírni az **append** függvényt kis lépésekben.

Láncolt lista megvalósítása

- Írjuk meg ehhez a C++ kódot. Azt szeretnénk, hogy ez a main függvény működjön:

```
1  int main(void) {
2    struct list_e *start = NULL; // pointer az első elemre
3    append(start, 1);            // hozzáadjuk az első elemet
4    append(start, 5);            // majd a másodikat
5    append(start, -2);           // és még egy harmadikat
6    for(;;) {    // valahogy végig kell majd menni a listán
7      // itt irassuk ki a lista elemeit
8    }
9    return 0;
10 }
```

- Kezdjük el megírni az **append** függvényt kis lépésekben.
- Mivel lépésekben dolgozunk, így lesznek pontok amikor hibás kód lesz a diákon, mindig felhívom majd a figyelmet, hogy miért hibás.

Láncolt lista megvalósítása

- Írjuk meg ehhez a C++ kódot. Azt szeretnénk, hogy ez a main függvény működjön:

```
int main(void) {
  struct list_e *start = NULL; // pointer az első elemre
  append(start, 1);            // hozzáadjuk az első elemet
  append(start, 5);            // majd a másodikat
  append(start, -2);           // és még egy harmadikat
  for(;;) {    // valahogy végig kell majd menni a listán
    // itt irassuk ki a lista elemeit
  }
  return 0;
}
```

- Kezdjük el megírni az **append** függvényt kis lépésekben.
- Mivel lépésekben dolgozunk, így lesznek pontok amikor hibás kód lesz a diákon, mindig felhívom majd a figyelmet, hogy miért hibás.
- Használjuk a pár diával korábban definiált **struct list_e** típust.

Láncolt lista megvalósítása

- Azzal kell kezdenünk, hogy az első elemet berakjuk, ekkor a kapott **start**-ban még **NULL** van, hisz még nincs elem amit tárolnánk.

```
1 void append(struct list_e *start, int n) {
2   struct list_e *e = new struct list_e;
3   (*e).num = n;
4   (*e).next = NULL;
5   start = e;
6 }
```

Láncolt lista megvalósítása

- Azzal kell kezdenünk, hogy az első elemet berakjuk, ekkor a kapott **start**-ban még **NULL** van, hisz még nincs elem amit tárolnánk.

```
1 void append(struct list_e *start, int n) {
2   struct list_e *e = new struct list_e;
3   (*e).num = n;
4   (*e).next = NULL;
5   start = e;
6 }
```

- Dinamikusan hozzuk létre az új elemet, hogy megmaradjon a függvény után is.

Láncolt lista megvalósítása

- Azzal kell kezdenünk, hogy az első elemet berakjuk, ekkor a kapott **start**-ban még **NULL** van, hisz még nincs elem amit tárolnánk.

```
1 void append(struct list_e *start, int n) {
2   struct list_e *e = new struct list_e;
3   (*e).num = n;
4   (*e).next = NULL;
5   start = e;
6 }
```

- Dinamikusan hozzuk létre az új elemet, hogy megmaradjon a függvény után is.
- Ha ezt kipróbáljuk csak 1 **append** függvény hívással a main-ben, akkor se lesz jó.

Láncolt lista megvalósítása

- Azzal kell kezdenünk, hogy az első elemet berakjuk, ekkor a kapott **start**-ban még **NULL** van, hisz még nincs elem amit tárolnánk.

```
1  void append(struct list_e *start, int n) {
2    struct list_e *e = new struct list_e;
3    (*e).num = n;
4    (*e).next = NULL;
5    start = e;
6  }
```

- Dinamikusan hozzuk létre az új elemet, hogy megmaradjon a függvény után is.
- Ha ezt kipróbáljuk csak 1 **append** függvény hívással a main-ben, akkor se lesz jó.
- Hisz a **start** pointert szeretnénk megváltoztatni, nem a tárolt értékét. Szóval a pointer pointerét kell átadnunk.

Láncolt lista megvalósítása

- Itt csak a pointer pointert javítottuk:

```
1  void append(struct list_e **start, int n) {
2    struct list_e *e = new struct list_e;
3    (*e).num = n;
4    (*e).next = NULL;
5    *start = e; // most már *start kell
6  }
```

Láncolt lista megvalósítása

- Itt csak a pointer pointert javítottuk:

```
1 void append(struct list_e **start, int n) {
2   struct list_e *e = new struct list_e;
3   (*e).num = n;
4   (*e).next = NULL;
5   *start = e; // most már *start kell
6 }
```

- Ez már működik, 1 elemet hozzá tudunk adni.

Láncolt lista megvalósítása

- Itt csak a pointer pointert javítottuk:

```
1  void append(struct list_e **start, int n) {
2    struct list_e *e = new struct list_e;
3    (*e).num = n;
4    (*e).next = NULL;
5    *start = e; // most már *start kell
6  }
```

- Ez már működik, 1 elemet hozzá tudunk adni.
- Viszont az `(*x).y` jelölés hamar fájdalmassá válhat, erre szerencsére van megoldás. Ekvivalens az `x->y` jelöléssel.

Láncolt lista megvalósítása

- Csak a `x->y` lett változtatva:

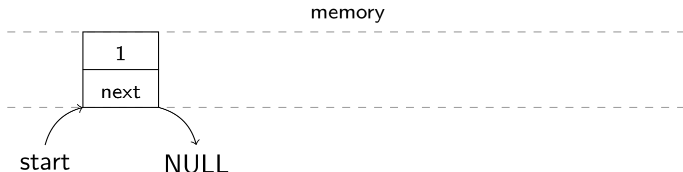
```
1  void append(struct list_e **start, int n) {
2    struct list_e *e = new struct list_e;
3    e->num = n;
4    e->next = NULL;
5    *start = e; // most már *start kell
6  }
```

Láncolt lista megvalósítása

- Csak a `x->y` lett változtatva:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  *start = e; // most már *start kell
}
```

- Most valahogy így néz ki a memóriánk:

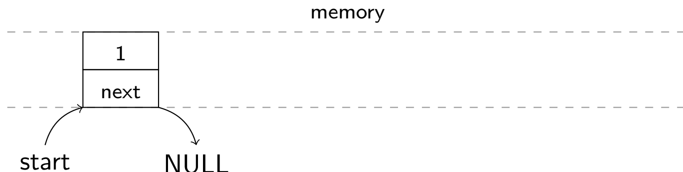


Láncolt lista megvalósítása

- Csak a `x->y` lett változtatva:

```
1  void append(struct list_e **start, int n) {
2    struct list_e *e = new struct list_e;
3    e->num = n;
4    e->next = NULL;
5    *start = e; // most már *start kell
6  }
```

- Most valahogy így néz ki a memóriánk:



- De ha lefuttatnánk mégegyszer az **append**-et (a második elemet megpróbálnánk berakni) könnyen látható, hogy az szimplán csak lecserélné az 1. elemet.

Láncolt lista megvalósítása

- Ágazzunk el aszerint, hogy az 1. elemet rakjuk-e be, vagy nem:

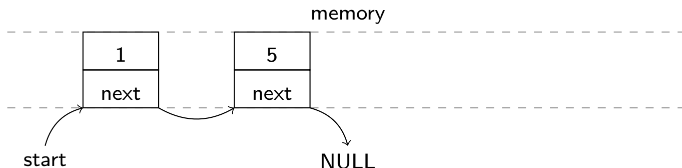
```
1  void append(struct list_e **start, int n) {
2    struct list_e *e = new struct list_e;
3    e->num = n;
4    e->next = NULL;
5    if (*start == NULL) {
6      *start = e;
7    } else {
8      (*start)->next = e;
9    }
10 }
```

Láncolt lista megvalósítása

- Ágazzunk el aszerint, hogy az 1. elemet rakjuk-e be, vagy nem:

```
void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e;
  } else {
    (*start)->next = e;
  }
}
```

- Most valahogy így néz ki a memóriánk (ha a 2. append megvolt):

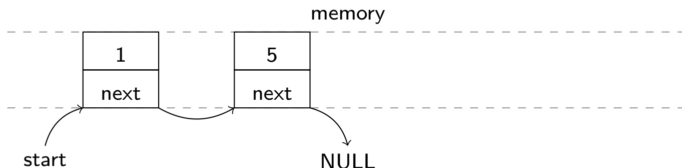


Láncolt lista megvalósítása

- Ágazzunk el aszerint, hogy az 1. elemet rakjuk-e be, vagy nem:

```
1  void append(struct list_e **start, int n) {
2    struct list_e *e = new struct list_e;
3    e->num = n;
4    e->next = NULL;
5    if (*start == NULL) {
6      *start = e;
7    } else {
8      (*start)->next = e;
9    }
10 }
```

- Most valahogy így néz ki a memóriánk (ha a 2. append megvolt):



- Ismét egyszerű látni, hogy ha megint futtatjuk az **append**-et, akkor mindig a második elemet cseréljük le, hisz a **start next**-jét módosítjuk.

Láncolt lista bejárása

- Most álljunk meg egy pillanatra és gondoljuk végig hogyan lehetne elnavigálni az utolsó elemhez, hisz mindig annak kellene a **next**-jét az új elem pointerére állítani.

Láncolt lista bejárása

- Most álljunk meg egy pillanatra és gondoljuk végig hogyan lehetne elnavigálni az utolsó elemhez, hisz mindig annak kellene a **next**-jét az új elem pointerére állítani.
- Ismét lophatunk a C string-ből, ott addig mentünk amíg a karakter nem a **'\0'** lezáró nulla. Most addig mehetünk amíg a pointer nem a **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Láncolt lista bejárása

- Most álljunk meg egy pillanatra és gondoljuk végig hogyan lehetne elnavigálni az utolsó elemhez, hisz mindig annak kellene a **next**-jét az új elem pointerére állítani.
- Ismét lophatunk a C string-ből, ott addig mentünk amíg a karakter nem a **'\0'** lezáró nulla. Most addig mehetünk amíg a pointer nem a **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

- Ha végiggondoljuk, ez egy tipikus **for** ciklus:

Láncolt lista bejárása

- Most álljunk meg egy pillanatra és gondoljuk végig hogyan lehetne elnavigálni az utolsó elemhez, hisz mindig annak kellene a **next**-jét az új elem pointerére állítani.
- Ismét lophatunk a C string-ből, ott addig mentünk amíg a karakter nem a **'\0'** lezáró nulla. Most addig mehetünk amíg a pointer nem a **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

- Ha végiggondoljuk, ez egy tipikus **for** ciklus:
  - Inicializáljuk a ciklusváltozót (**e**) az első elemmel.

Láncolt lista bejárása

- Most álljunk meg egy pillanatra és gondoljuk végig hogyan lehetne elnavigálni az utolsó elemhez, hisz mindig annak kellene a **next**-jét az új elem pointerére állítani.
- Ismét lophatunk a C string-ből, ott addig mentünk amíg a karakter nem a **'\0'** lezáró nulla. Most addig mehetünk amíg a pointer nem a **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

- Ha végiggondoljuk, ez egy tipikus **for** ciklus:
  - Inicializáljuk a ciklusváltozót (**e**) az első elemmel.
  - C string-es megállási feltétel.

Láncolt lista bejárása

- Most álljunk meg egy pillanatra és gondoljuk végig hogyan lehetne elnavigálni az utolsó elemhez, hisz mindig annak kellene a **next**-jét az új elem pointerére állítani.
- Ismét lophatunk a C string-ből, ott addig mentünk amíg a karakter nem a **'\0'** lezáró nulla. Most addig mehetünk amíg a pointer nem a **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

- Ha végiggondoljuk, ez egy tipikus **for** ciklus:
  - Inicializáljuk a ciklusváltozót (**e**) az első elemmel.
  - C string-es megállási feltétel.
  - Lépünk a következő elemre.

Láncolt lista bejárása

- Most álljunk meg egy pillanatra és gondoljuk végig hogyan lehetne elnavigálni az utolsó elemhez, hisz mindig annak kellene a **next**-jét az új elem pointerére állítani.
- Ismét lophatunk a C string-ből, ott addig mentünk amíg a karakter nem a **'\0'** lezáró nulla. Most addig mehetünk amíg a pointer nem a **NULL** pointer:

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

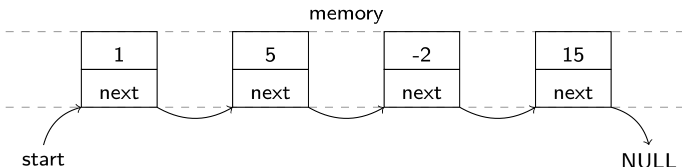
- Ha végiggondoljuk, ez egy tipikus **for** ciklus:
  - Inicializáljuk a ciklusváltozót (**e**) az első elemmel.
  - C string-es megállási feltétel.
  - Lépünk a következő elemre.
- Ezt használhatjuk is a **main**-ben a kiíratásra.

Láncolt lista bejárása

Nézzük meg hogy nézne ki ez a ciklus bejárás a (még csak elképzelt) végleges láncolt listánkon.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Hol tartunk: ciklus előtt



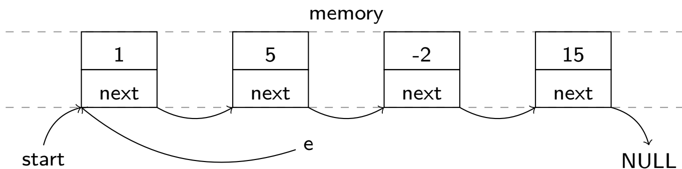
Kiemenet:

Láncolt lista bejárása

Nézzük meg hogy nézne ki ez a ciklus bejárás a (még csak elképzelt) végleges láncolt listánkon.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Hol tartunk: inicializálás megtörtént



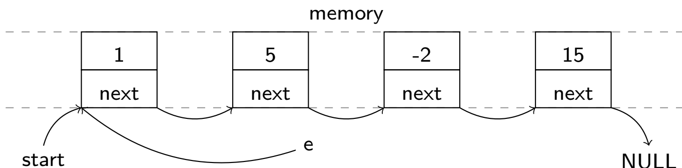
Kiemenet:

Láncolt lista bejárása

Nézzük meg hogy nézne ki ez a ciklus bejárás a (még csak elképzelt) végleges láncolt listánkon.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Hol tartunk: első ciklus vége



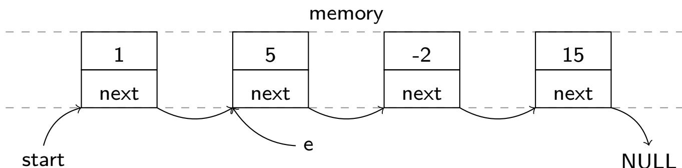
Kiemenet:
1

Láncolt lista bejárása

Nézzük meg hogy nézne ki ez a ciklus bejárás a (még csak elképzelt) végleges láncolt listánkon.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Hol tartunk: második ciklus eleje



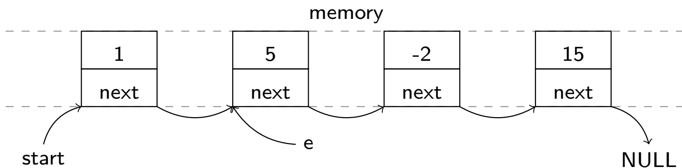
Kiemenet:
1

Láncolt lista bejárása

Nézzük meg hogy nézne ki ez a ciklus bejárás a (még csak elképzelt) végleges láncolt listánkon.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Hol tartunk: második ciklus vége



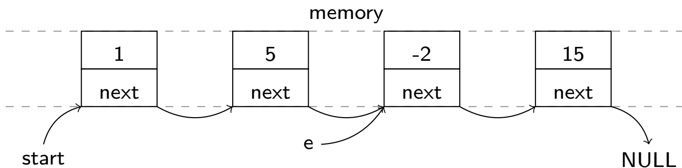
Kiemenet:
1
5

Láncolt lista bejárása

Nézzük meg hogy nézne ki ez a ciklus bejárás a (még csak elképzelt) végleges láncolt listánkon.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Hol tartunk: harmadik ciklus vége



Kiemenet:

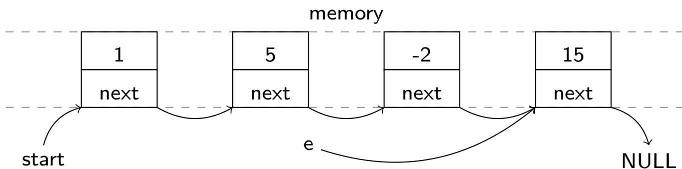
1
5
-2

Láncolt lista bejárása

Nézzük meg hogy nézne ki ez a ciklus bejárás a (még csak elképzelt) végleges láncolt listánkon.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Hol tartunk: negyedik ciklus vége



Kiemenet:

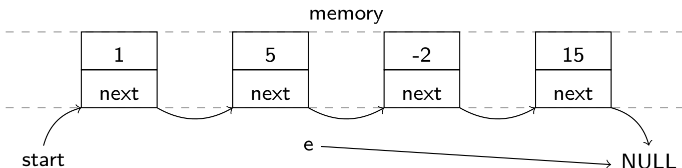
1
5
-2
15

Láncolt lista bejárása

Nézzük meg hogy nézne ki ez a ciklus bejárás a (még csak elképzelt) végleges láncolt listánkon.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Hol tartunk: negyedik ciklus vége és megtörtént a léptetés



Kiemenet:

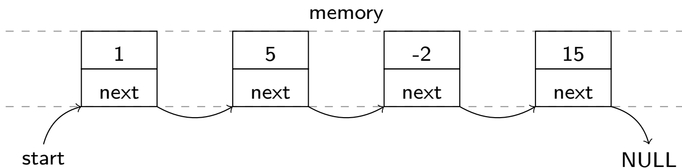
1
5
-2
15

Láncolt lista bejárása

Nézzük meg hogy nézne ki ez a ciklus bejárás a (még csak elképzelt) végleges láncolt listánkon.

```
for(struct list_e *e = start; e != NULL; e = e->next) {
  cout << e->num << endl;
}
```

Hol tartunk: ciklus vége, nem teljesült a feltétel



Kiemenet:

1
5
-2
15

Láncolt lista megvalósítása

- Térjünk vissza a megvalósításra és használjuk a tanultakat:

```
1  void append(struct list_e **start, int n) {
2    struct list_e *e = new struct list_e;
3    e->num = n;
4    e->next = NULL;
5    if (*start == NULL) {
6      *start = e; // most már *start kell
7    } else {
8      struct list_e *p = NULL;
9      for(p = *start; p->next != NULL; p = p->next){}
10     p->next = e;
11   }
12 }
```

Láncolt lista megvalósítása

- Térjünk vissza a megvalósításra és használjuk a tanultakat:

```
1  void append(struct list_e **start, int n) {
2    struct list_e *e = new struct list_e;
3    e->num = n;
4    e->next = NULL;
5    if (*start == NULL) {
6      *start = e; // most már *start kell
7    } else {
8      struct list_e *p = NULL;
9      for(p = *start; p->next != NULL; p = p->next){}
10     p->next = e;
11   }
12 }
```

- A ciklus amit használunk kicsit eltér a kiírásnál használt bejárás ciklusától.

Láncolt lista megvalósítása

- Térjünk vissza a megvalósításra és használjuk a tanultakat:

```
1   void append(struct list_e **start, int n) {
2     struct list_e *e = new struct list_e;
3     e->num = n;
4     e->next = NULL;
5     if (*start == NULL) {
6       *start = e; // most már *start kell
7     } else {
8       struct list_e *p = NULL;
9       for(p = *start; p->next != NULL; p = p->next){}
10      p->next = e;
11    }
12  }
```

- A ciklus amit használunk kicsit eltér a kiírásnál használt bejárás ciklusától.
- Akkor szeretnénk megállni amikor az utolsó elemnél vagyunk.

Láncolt lista megvalósítása

- Térjünk vissza a megvalósításra és használjuk a tanultakat:

```
1  void append(struct list_e **start, int n) {
2    struct list_e *e = new struct list_e;
3    e->num = n;
4    e->next = NULL;
5    if (*start == NULL) {
6      *start = e; // most már *start kell
7    } else {
8      struct list_e *p = NULL;
9      for(p = *start; p->next != NULL; p = p->next){}
10     p->next = e;
11   }
12 }
```

- A ciklus amit használunk kicsit eltér a kiírásnál használt bejárás ciklusától.
- Akkor szeretnénk megállni amikor az utolsó elemnél vagyunk.
- Onnan tudjuk, hogy az utolsó elemnél vagyunk, hogy az ő **next**-je már **NULL**.

Láncolt lista megvalósítása

- Térjünk vissza a megvalósításra és használjuk a tanultakat:

```
1  void append(struct list_e **start, int n) {
2    struct list_e *e = new struct list_e;
3    e->num = n;
4    e->next = NULL;
5    if (*start == NULL) {
6      *start = e; // most már *start kell
7    } else {
8      struct list_e *p = NULL;
9      for(p = *start; p->next != NULL; p = p->next){}
10     p->next = e;
11   }
12 }
```

- A ciklus amit használunk kicsit eltér a kiírásnál használt bejárás ciklusától.
- Akkor szeretnénk megállni amikor az utolsó elemnél vagyunk.
- Onnan tudjuk, hogy az utolsó elemnél vagyunk, hogy az ő **next**-je már **NULL**.
- A cikluson belül nem csinálunk semmit, nekünk csak az fontos, hogy a **p** pointerrel az utolsó elemre lépjünk.

Láncolt lista megvalósítása

- Térjünk vissza a megvalósításra és használjuk a tanultakat:

```
1   void append(struct list_e **start, int n) {
2     struct list_e *e = new struct list_e;
3     e->num = n;
4     e->next = NULL;
5     if (*start == NULL) {
6       *start = e; // most már *start kell
7     } else {
8       struct list_e *p = NULL;
9       for(p = *start; p->next != NULL; p = p->next){}
10      p->next = e;
11    }
12  }
```

- A ciklus amit használunk kicsit eltér a kiírásnál használt bejárás ciklusától.
- Akkor szeretnénk megállni amikor az utolsó elemnél vagyunk.
- Onnan tudjuk, hogy az utolsó elemnél vagyunk, hogy az ő **next**-je már **NULL**.
- A cikluson belül nem csinálunk semmit, nekünk csak az fontos, hogy a **p** pointerrel az utolsó elemre lépjünk.
- Ez már valóban megvalósít egy memória határáig dinamikusan bővülő adatszerkezetet.

Láncolt lista megvalósítása (teljes kód)

```
#include <iostream>

using namespace std;

struct list_e {
  int num;
  struct list_e *next;
};

void append(struct list_e **start, int n) {
  struct list_e *e = new struct list_e;
  e->num = n;
  e->next = NULL;
  if (*start == NULL) {
    *start = e; // most már *start kell
  } else {
    struct list_e *p = NULL;
    for(p = *start; p->next != NULL; p = p->next){}
    p->next = e;
  }
}
```

```
int main(void) {
  struct list_e *start = NULL;
  append(&start, 1);
  append(&start, 5);
  append(&start, -2);
  append(&start, 15);
  for(struct list_e *e = start; e != NULL; e = e->next) {
    cout << e->num << endl;
  }
  return 0;
}
```

Továbbiakban

Hova tovább?

Továbbiakban

Hova tovább?

- A megvalósított láncolt listánkat felszabadítani (törölni) is tudnunk kellene.

Továbbiakban

Hova tovább?

- A megvalósított láncolt listánkat felszabadítani (törölni) is tudnunk kellene.
- Hosszútávon kezelhetetlen lenne, ha mindig **append** és hasonló függvényeket használnánk (kellene egy append_list, append_dictionary, append_set, stb.). Erre lesznek megoldások az osztályok (**class**).

Továbbiakban

Hova tovább?

- A megvalósított láncolt listánkat felszabadítani (törölni) is tudnunk kellene.
- Hosszútávon kezelhetetlen lenne, ha mindig **append** és hasonló függvényeket használnánk (kellene egy append_list, append_dictionary, append_set, stb.). Erre lesznek megoldások az osztályok (**class**).
- Nehézséget okoz az is, hogy bele van égetve az adatszerkezetbe, hogy mit tárol (a mi esetünkben 1 **int**-et). Erre lesz megoldás a **template**.

Továbbiakban

Hova tovább?

- A megvalósított láncolt listánkat felszabadítani (törölni) is tudnunk kellene.
- Hosszútávon kezelhetetlen lenne, ha mindig **append** és hasonló függvényeket használnánk (kellene egy append_list, append_dictionary, append_set, stb.). Erre lesznek megoldások az osztályok (**class**).
- Nehézséget okoz az is, hogy bele van égetve az adatszerkezetbe, hogy mit tárol (a mi esetünkben 1 **int**-et). Erre lesz megoldás a **template**.
- Nem tarthatjuk ezeket mind egy file-ban, ha egyre több ilyet használunk. Nemsokára elkezdünk több forrás file-al dolgozni és **header** file-okkal.

Továbbiakban

Hova tovább?

- A megvalósított láncolt listánkat felszabadítani (törölni) is tudnunk kellene.
- Hosszútávon kezelhetetlen lenne, ha mindig **append** és hasonló függvényeket használnánk (kellene egy append_list, append_dictionary, append_set, stb.). Erre lesznek megoldások az osztályok (**class**).
- Nehézséget okoz az is, hogy bele van égetve az adatszerkezetbe, hogy mit tárol (a mi esetünkben 1 **int**-et). Erre lesz megoldás a **template**.
- Nem tarthatjuk ezeket mind egy file-ban, ha egyre több ilyet használunk. Nemsokára elkezdünk több forrás file-al dolgozni és **header** file-okkal.
- Nem implementálhatjuk mindig ezeket magunk. Erre lesznek majd a már létező könyvtárak, melyekben mindezek már implementálva vannak.

kisZH kérdések

- Rajzoljuk le egy láncolt lista memória képét vázlatosan!
- Mi az a **NULL** pointer?
- Mutassunk példát egy dinamikus tömb létrehozására!
- Mire való a **delete** és a **delete[]**?
- Mi a kedvenc állatotok?