Láncolt lista

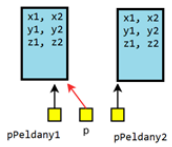
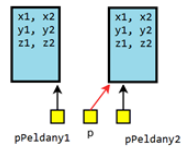
1. bevezetés, dinamikus memória foglalás

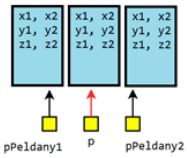
Hozzunk létre olyan struktúrát, amely képes a 2 dimenziós képernyőn lévő pontok alapján egy háromszög koordinátáit tárolni. A példányosításkor jönnek létre az objektumok, vagyis azok a memória területek amik a 6 koordinátát tárolják egységbezárva. Az objektum számára szükséges memória területet pointer segítségével dinamikusan lefoglaljuk le.

```
#include <iostream>
using namespace std;
struct triangle
{
        unsigned int x1, x2, y1, y2, z1, z2;//összesen 6 koordináta
};
int main()
{
        setlocale(LC_ALL, "hun");
        //példányosítás - instantiation
        triangle peldany1;
        triangle peldany2;

        //dinamikus példányosítás
        triangle* pPeldany1;
        triangle* pPeldany2;
        triangle* p;
        //3 pointert deklaráltam ami olyan objektumra képes mutatni aminek a típusa triangle
        pPeldany1 = new triangle;
        pPeldany2 = new triangle;
        //------------------------

        p = pPeldany1;
        p = pPeldany2;
        p = new triangle;
        p->x1 = 10;
        system("pause");
        return 0;
}
```

A pointer neve elé szokás egy **p** betűt írni, ez abban segít, hogy a változó nevéből már közvetlenül kiderül, hogy az pointer típusú.

A kék téglalap egy háromszög típusú objektumot ábrázol. Látható, hogy az értékadás segítségével a **pPeldany1** pointer rámutat egy olyan memória területre (objektumra) ahol el vannak tárolva a háromszög koordinátái.

	1. A **p** mutató már nem foglal le újabb területet csak rámutat oda ahova **pPeldany1** is mutat.
	2. Aztán **p**-vel rámutatunk a jobb oldali háromszögre (objektumra), oda ahova **pPeldany2** is mutat.

	3. Majd **p**-vel rámutatunk egy újabb objektumra.

A rámutatás azt jelenti, hogy p -ben tároljuk a mutatott objektum címét. Jegyezzük meg, hogy jelen esetben a háromszög csúcsainak koordinátáit az objektumhoz hozzá tartozó tagváltozókban tároltuk. Az értékadás pointeren keresztül a nyíl operátorral válik lehetségessé.

```
p->x1 = 10;
```

2. statikus és dinamikus tömbök

A háromszög csúcsainak koordinátáit lehet ún. statikus tömbben tárolni.

```
unsigned int koordinataTomb[6];
```

Az előjel nélküli 6 elemű integer értékeket tartalmazó tömb a struktúra tagváltozója. Így az értékadás kétféleképpen lehetséges.

```
#include <iostream>
using namespace std;
struct triangle
{
	unsigned int koordinataTomb[6];
};
int main()
{
	setlocale(LC_ALL, "hun");
	//példányosítás - instantiation
	triangle peldany1;
	peldany1.koordinataTomb[1] = 10;
	//dinamikus:
	triangle* pPeldany1 = new triangle;
	pPeldany1->koordinataTomb[1] = 10;

	system("pause");
	return 0;
}
```

1. pont operátort használunk normál értékadásnál:

```
peldany1.koordinataTomb[1] = 10;
```

2. nyíl operátort kell használni ha a tagelérés pointeren keresztül megy végbe indirekt módon.

```
pPeldany1->koordinataTomb[1] = 10;
```

Ha nem egy háromszög koordinátáit kell tárolni hanem egy tetszőleges oldalú sokszögét akkor használhatunk dinamikus tömböt is. A dinamikus tömb előnye, hogy a méret nem fordítási időben derül ki, hanem futási időben (runtime). Ezért van az, hogy a tömb méretét nem kell tudnunk előre, elegendő ha a program futásakor a felhasználó adja meg azt.

```
#include <iostream>
using namespace std;
struct triangle
{
	int size;
	int getSize();
	void makeArray(int* p1, int size);
};
int main()
{
	setlocale(LC_ALL, "hun");
	triangle p1;
	int meret = p1.getSize();
	int* pTomb = new int[meret];
	p1.makeArray(pTomb, meret);
	system("pause");
	return 0;
}
int triangle::getSize()
{
	cout << "Kérem adja meg sokszög csúcsainak számát!" << endl;
	cin >> size;
	return size * 2;//minden csúcshoz két koordináta tartozik
}
void triangle::makeArray(int* p1, int size)
{
	//feltöltés véletlen számokkal 1-10 között
	int i = 0;
	while (i<size)
	{
		p1[i] = rand() % 10;
		i++;
	}
	i = 0;
	while (i<size)
	{
		cout << i + 1 <<". koordináta: "<<p1[i] << endl;
		i++;
	}
}
```

A tömbök előnye: gyors, közvetlen adatelérés. Az elemeket tetszőleges sorrendben, közvetlen címzéssel érjük el, hiszen közvetlenül egymás mellett helyezkednek el a memóriában. Ezt gyakran ki is használjuk pl. cserés rendezés esetén, amikor egymás mellett lévő elemeket megcseréljük ha a sorrendjük nem megfelelő.

A tömbök hátránya: lassú az átméretezés. Mivel feltétlenül egymás mellett kell, hogy legyenek az elemek, ha változtatni akarjuk a tömb méretét, újra kell foglalni a memóriaterületet.

ennek lépései:

(I.) Le kell foglalni másutt a szükséges méretű területet

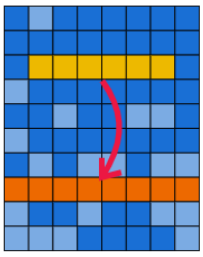
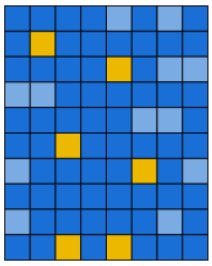
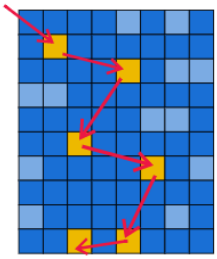
(II.) Át kell másolni az elemeket a régi helyről

(III.) Fel kell szabadítani a régi tömböt.

A tömbök dinamikus nyújtása ezért nagyon költséges művelet. Ráadásul másolás közben az eredeti tartalom kétszer szerepel a memóriában! A tömböket nem érdemes használni pl.: a következő feladat megoldásához sem. **Kezeljük egy szerverre bejelentkezett felhasználók listáját!**

- Nem tudhatjuk, hogy hányan akarnak majd bejelentkezni hozzánk.
- A felhasználók száma folyamatosan változik.

Ideális megoldás lenne: minden belépéskor csak az új felhasználónak megfelelő területet foglalni. Ezzel az a probléma, hogy a memóriában elszórva helyezkednek el az adatok. Valahogyan nyilván kellene tartani, hogy hol vannak az egyes elemek! Ha ehhez pointerek tömbjét használnánk, akkor az lesz az, amit folyton át kell méretezni – vagyis visszakapnánk az eredeti problémát.

 tömb a memóriában, nagyon lassú az átméretezés	 egyesével foglalt memóriaterületek	 egyesével foglalt memóriaterületek **láncolva**

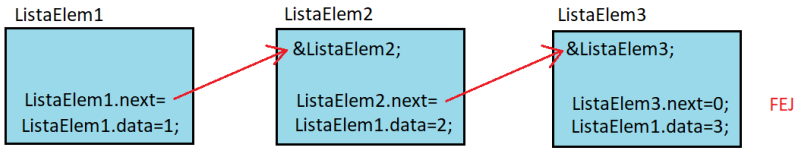
Ötlet: az egyes elemek tárolják az őket követő elem címét! Így minden elem egyesével foglalható. Minden elem adatát kiegészítjük egy mutatóval, ami ugyan plusz költség, de egy olyan adatszerkezetet kapunk, amire teljesül, hogy:

1. tetszőleges méretűre bővíthető dinamikusan,
2. új elem hozzáadásának a költsége nagyon kicsi,
3. elemek törlése is olcsó művelet.

Vegyük észre, hogy minderre a dinamikus memóriakezelés ad lehetőséget: egy olyan adatszerkezetet készülünk most létrehozni, amelyben az egyes elemek külön jönnek létre, és külön szűnhetnek meg.

3. A legegyszerűbb láncolt lista 3 elemmel

Láncolt lista (linked list) olyan adatszerkezet, ahol az egyes elemek (node) láncba vannak fűzve azáltal, hogy tárolják a következő elem címét. Nyelvi szinten egy láncolt listába fűzhető elem egy struktúrával írható le:




```
struct ListaElem
{
        string adat;
        ListaElem* next;
};
```

```
#include <iostream>
#include <string>
using namespace std;
struct ListaElem
{
	int data;
	ListaElem* next;
};
int main()
{
	setlocale(LC_ALL, "hun");
	//fűzzünk össze 3 elemet és töltsük fel 3 egész számmal 1,2,3
	ListaElem elem1;
	ListaElem elem2;
	ListaElem elem3;

	elem1.data = 1;
	elem1.next = &elem2;

	elem2.data = 2;
	elem2.next = &elem3;

	elem3.data = 3;
	elem3.next = NULL;


	ListaElem* i = &elem1;
	while (i != NULL)
	{
		cout << i->data << endl;
		i = i->next;
	}
	cout << "---" << endl;

	system("pause");
	return 0;
}
```

Lépések:

1. hozzunk létre struktúra példányokat
2. a data tagváltozónak adjunk értéket
3. a next pointer tagváltozót állítsuk rá következő objektum címére

Jegyezzük meg, hogy a tagváltozóban tárolt cím a következő objektum címe.

4. A lista felépítése függvénnyel

Azért, hogy az elemek létrehozása ne egyenként valósuljon meg, érdemes függvényt használni. Egy olyan függvényt érdemes megírni amelyik a következő feladatokat képes végrehajtani:

1. példányosítson egy új lista elemet, ez lesz a lista feje, ezt mindig NULL értékre állítjuk
2. az utólsó elem a **fej**
3. a példányosítás legyen dinamikus, így példányoknak nincs neve csak címe (ez azért fontos, mert ha nincs a példánynak neve az most előny számunkra hiszen nem kell nevet adni)
4. jusson be a függvénybe az előző listaelem
5. az előző listaelem tagpointerében tároljuk el az új listaelem címét
6. az új listaelembe helyezzük el a tárolni kívánt adatot
7. juttassuk ki, azaz térjünk vissza az új lista elem címével, hiszen a következő függvényhívásnál már ez a cím fog az előző elemnek számítani

A FEJ elem a függvényen kívül jön létre

```
#include <iostream>
using namespace std;
struct ELEM
{
	int data;
	ELEM* next;
};
ELEM* addElement(ELEM* elozo, int data)
{
	ELEM* ujElem = new ELEM;
	ujElem->data = data;
	ujElem->next = NULL;
	//-----------------
	elozo->next = ujElem; //összekapcsolás
	return ujElem;
}
int main()
{
	setlocale(LC_ALL, "hun");
	ELEM* mozgo = new ELEM;
	ELEM* elso;
	int szam = 0;
	cout << "Kérem adja a tárolni kívánt értéket!" << endl;
	cin >> szam;
	mozgo->data = szam;
	mozgo->next = NULL;
	elso = mozgo;//mielőtt a mozgó pointer értékét megváltoztatjuk, el mentjük az
	//értékét az első nevű pointerben

	//ez a lépés biztosítja, hogy a pointerben lévő korábbi érték megmarad
	//annak ellenére, hogy a címét mozgatjuk a következő elemre, azért a neve mozgo

	//feltöltés, végjelig
	while (szam!=0)
	{
		cin >> szam;
		mozgo = addElement(mozgo, szam);
	}
	cout << "---" << endl;
	//kiíratás
	ELEM* i = elso;
	while (i != NULL)
	{
		cout << i->data << endl;
		i = i->next;
	}
	cout << "---" << endl;
	ELEM* tmp;
	i = elso;
	while (i != NULL)
	{
		tmp = i;
		i = i->next;
		delete tmp;
	}
	system("pause");
	return 0;
}
```