თემა 2. მეხსიერების დინამიკური მართვა, new, delete და delete[] ოპერატორები

საკითხები:

მეხსიერების დინამიკური მართვის აუცილებლობა.

აქამდე, განხილულ ამოცანებში ჩვენს მიერ შექმნილი ყველა ობიექტი (ცვლადები, კონტეინერები, კლასების წევრები) არსებობას წყვეტდნენ ან პროგრამის დასრულების, ან იმ პროგრამული ბლოკის დახურვის შემდეგ სადაც მათზე გაკეთდა განაცხადი.

ასეთი მიდგომა მხოლოდ მარტივ პატარა პროგრამებში ამართლებს. რეალურ ამოცანებში, მრავალი სხვადასხვა მიზეზის გამო ობიექტი იქმნება მაშინ, როდესაც ამის აუცილებლობა ჩნდება, ხოლო მის მიერ დაკავებული მეხსიერება თავისუფლდება მაშინ, როდესაც ობიექტის გამოყენება მთავრდება. თუ არ ვიზრუნებთ დაკავებული მეხსიერების სისტემისთვის დაბრუნებაზე, ანუ გათავისუფლებაზე, შეიქმნება პრობლება მეხსიერებაზე ახალი განაცხადების დაკმაყოფილებასთან დაკავშირებით, ე.წ. "მეხსიერების გაჟონვა" (memory leak).

C++ ენა გვაძლევს მეხსიერების დინამიკური მართვის შესაძლებლობას. ზოგიერთი თანამედროვე ენა (მაგალითად Java) იყენებებს განსაკუთრებულ მექანიზმს (ე.წ. ნარჩენების შემგროვებელი პროგრამა), რომელიც ზრუნავს რომ მეხსიერება არ გადაივსოს ისეთი ობიექტებისთვის გამოყოფილი ადგილებით, რომლებსაც პროგრამა აღარ იყენებს. ეს პროგრამისტისთვის მოსახერხებელია და საიმედო, თუმცა საზღაურს წარმოადგენს კომპრომისი სწრაფქმედებაში. თანამედროვე C++ გვთავაზობს განსხვავებულ მიდგომას, რომელსაც ჩვენ ვისწავლით სემესტრის ბოლო მეცადინეობებზე, ხოლო უფრო სრულად - გაღრმავებული არჩევითი კურსის ფარგლებში.

სიტყვა dynamic, განმარტების თანახმად, ნიშნავს ზედსართავის შემთხვევაში" (of a process or system) characterized by constant change, activity, or progress, ხოლო სახელის შემთხვევაში: a force that stimulates change or progress within a system or process. ნებისმიერ შემთხვევაში ,დინამიკა დაკავშირებულია ცლვილებასა და განვითარებასთან.

დინამიკური მეხსიერების უკეთ გაგებისთვის მოკლედ შევეხოთ C++ პროგრამის მეხსიერების სახეებს. პროგრამას შეუძლია ორი სახის მეხსიერების გამოყენება:

- **სტეკი** - ფუნქციებში შექმნილი ყველა ცვადი და ობიექტი იყენებს ეხსიერებას სტეკიდან.
- **გროვა** - ეს მეხსიერება შეიძლება გამოყენებულ იქნას დინამიკურად, როდესაც პროგრამა გაშვებულია.

შევნიშნოთ, რომ მეხსიერების სტეკსა და გროვას არაფერი აქვთ საერთო იმ მონაცემთა სტრუქტურებთან, რომლებსაც ჰქვია სტეკი და გროვა.

ერთი მიზეზი დინამიკური მეხსირების გამოყენებისთვის დასაწყისშივე აღვნიშნეთ - როცა აღარ დაგვჭირდება ეს მეხსიერება, დავუბრუნებთ სისტემას. მეორე მიზეზი ისაა, რომ მშირად წინასწარ არ ვიცით თუ რა მოცულობის მეხსიერება დაგვჭირდება , ამას ვიგებთ პროგრამის მსვლელობის დროს და, შესაბამისად, ვიყენებთ ამ მომენტისთვის ხელმისაწვდომ მეხსიერებას.

<<< new, delete და delete[] ოპერატორები ცვლადებთან და ობიექტებთან.

ოპერატორ new -ს უშუალოდ მოჰყვება მონაცემთა ტიპის სახელი და დაბრუნდება ამ ტიპის ახლადშექმნილი ობიექტის მისამართი. მონაცემთა ტიპს, თავის მხრივ, უნდა მოჰყვებოდეს რაიმე ტიპის კონსტრუქტორი, პარამეტრიანი ან უპარამეტრო. თუ კონსტრუქტორი პარამეტრიანია, new ახალგამოყოფილ მეხსიერებაში ჩაწერს მნიშვნელობას კონსტრუქტორის საშუალებით.

მაგალითად, `cout << *(new int(11)) << endl;` შეტყობინება დაბეჭდავს 11-ს. ”*” -ის მითითება აუცილებელია, რადგან ამის გარეშე დაიბეჭდება მისამართი თექვსმეტობითში.

თუ გვინდა new ოპერატორით შექმნილი ახალი ობიექტის მრავალჯერადი გამოყენება, აუცილებელია მისი მისამართი დავიმახსოვროთ შესაბამისი ტიპის პოინტერში. თუ გვინდა new ოპერატორით შექმნილი ობიექტის მიერ დაკავებული მეხსიერების გათავისუფლება, ანუ სისტემისთვის დაბრუნება, უნდა გამოვიყენოთ delete ოპერატორი, რომლის შემდეგაც დაიწერება ამ მეხსიერების დასაწყისის მისამართი, როგორც წესი იმ პოინტერის მითითებით რომელშიც იგი იქნა შენახული. მეხსიერების გათავისუფლების შემდეგ სასურველია რომ პოინტერს მიენიჭოს `nullptr` მნიშვნელობა, რათა შეცდომით არ მივაკითხოთ უკვე გათავისუფლებულ მეხსიერების მონაკვეთს.

განვიხილოთ მარტივი მაგალითი:

```
auto p = new pair<string, int>("SANGU", 2020);
cout << p->first << "  " << p->second << endl;
delete p;
p = nullptr;
```

მეხსიერების გაუქმების შემდეგ პოინტერს ძველი მნიშვნელობა რჩება, მაგრამ იქ აღარაა ის მონაცემები, რაც იყო (შემთხვევით, თუ ბევრი არაფერი შეიცვალა, შეიძლება იყოს კიდეც). ასეთ პოინტერს ჰქვია ველური პოინტერი (wild pointer). ხშირად, პროგრამისტებს ავიწყდებათ რომ უკვე გააუქმეს დაჯავშნული მეხსიერება და ისევ ცდილობენ შესვლას. საკმაოდ შარიანი შეცდომაა და ძნელად აღმოსაჩენიც. ამისგან მარტივი დაცვა არსებობს. გაუქმებას უნდა მოვაყოლოთ განულება და სადმე შეღწევის თეორიული შანსიც გამოირიცხება.

new ოპერატორი მუშაობს ნებისმიერი კლასის ნებისმიერ კონსტრუქტორთან, იქნება ეს ბიბლიოთეკის კლასი თუ მომხმარებლის მიერ შექმნილი. მაგალითად, თუ ერთ-ერთ წინა ლექციაზე შექმნილ ტბების კლასს (ოთხი კერძო ველით) დავუმატებთ კონსტრუქტორს

```
lake(ifstream & is);
```

იმპლემენტაციით

```
lake::lake(ifstream & is)
{
	is >> name >> area >> height >> depth;
}
```

მაშინ პროგრამა-დრაივერში შეგვიძლია გავაკეთოთ განაცხადი

```
	lake *p = new lake(ifs);
```

შემდეგ ვიმუშავებთ ამ ობიექტთან, რომლის ველები შეივსება `ifs` ნაკადით, შემდეგ გავაუქმებთ დაჯავშნილ მეხსიერებას და აგრეთვე პოინტერის მნიშვნელობას:

```
	delete p;
	p = nullptr;
```

C++ ენაში ნებისმიერი კონტეინერი კლასის ობიექტს წარმოადგენს. ამიტომ მათი დინამიკური შექმნა და გაუქმება მარტივია. მაგალითად:

```
auto p = new vector<int>({1,2,3,4,3,2,1});
for (auto e : (*p))
    cout << e << "  ";
cout << endl;
delete p;
p = nullptr;
```

<<< New და delete[] ოპერატორები ერთგანზომილებიან მასივებთან

ამ ოპერატორების გამოყენება ძალიან ეფექტურია დინამიკურ მასივებთან, ამიტომ ძირითადად ამ საკითხზე გავამახვილებთ ყურადღებას. შემდეგ მეცადინეობებზე კიდევ სხვა მაგალითებს შევხვდებით.

მასივი არის უმარტივესი კონტეინერი, რომელსაც აქვს ფიქსირებული ზომა და არვითარი მეთოდები (წევრი-ფუნქციები) არ გააჩნია. დინამიკური მასივი ფუნქციაში პარამეტრად გადაცემისას აუცილებლად კარგავს თავის ზომას (ჩვეულებრივი მასივისგან გასხვავებით, არ შველის ფუნქციის თარგის გამოყენება). მიუხედავად ამისა, მისი გამოყენება გამართლებულია როდესაც მთავარი საკითხი არის სისწრაფე.

ჩვეულებრივი ერთგანზომილებიანი მასივი არის ერთი და იმავე ტიპის ობიექტების კოლექცია, რომელიც მეხსიერებაში იკავებს ერთ მთლიან ფრაგმენტს. განაცხადი კეთდება შემდეგნაირად:

მასივის სახელი ობიექტის ტიპი [ობიექტების რაოდენობა];

მაგალითად:

```
int a[35];
Lake lk[10];
```

და ა.შ.

განაცხადის გაკეთების მომენტში შესაძლებელია მასივის ინიციალიზება:

```
int a[3] = {323,43,-34};
```

ან

```
int a[] = {3,443,-34, 90 55};
```

როგორც აღვნიშნეთ, ამგვარად შექმნილი მასივები არსებობენ ვიდრე არ დაიხურება ის პროგრამული ბლოკი, რომლის შიგნითაც გაკეთდა მათზე განაცხადი.

თუ მოვინდომებთ რომ ჩვენს მიერ შექმნილი მასივი იმართებოდეს დინამიკურად, მაშინ განაცხადი უნდა გავაკეთოთ T ტიპის n ეგზელმპლიარისგან შექმნილ **დინამიკურ** მასივზე

```
T *p = new T[n];
```

მაგალითად

```
int* mas = new int[20];
```

ქმნის მთელი რიცხვების 20 ელემენტიან მასივს სახელად mas. მასივის სახელი ბუნებრივად გაიგივდება მასივის, ანუ მის თავში მოთავსებული ელემენტის მისამართთან. მართლაც, მასივის სახელი ამ განმარტების თანახმად არის პოინტერი. ნებისმიერი q პოინტერის ირიბი მნიშვნელობა ანუ *q იგივეა რაც q[0], ხოლო პოინტერების არითმეტიკის თანახმად შემდეგი ელემენტების მნიშვნელობები იქნება *(q+1) ანუ q[1], და ა. შ.. ვიდრე არ შეიქმნება დინამიკური მასივის გაუქმების საჭიროება, იგი აღარაფრით განსხვავდება ჩვეულებრივი მასივისგან.

<<< მაგალითი 1. დავბეჭდოთ რაიმე T ტიპის მასივის ელემენტები ახალ-ახალ სტრიქონზე.

```
#include<iostream>
```

```
using namespace std;

template<typename T>
void printMassive(T* name, int size)
{
	for (int i = 0; i<size; i++)
		cout << name[i] << endl;
}
int main()
{
	int* mas = new int[20];
	for (int i = 0; i<20; i++)
		mas[i] = rand();
	printMassive(mas, 2);
	delete[] mas;
	mas = nullptr;
}
```

როგორც ვხედავთ, ფუნქციას მეორე პარამეტრად გადაეცემა დასაბეჭდი ელემენტების რაოდენობა, რადგან მასივის სახელმა არ იცის ელემენტების რაოდენობა. გამოძახებაში, უნდა მივაქციოთ ყურადღება რომ უფრო მეტის დაბეჭდვა არ მოვინდომოთ. ამ მარტივ შემთხვევაში ბევრი არაფერი გააფუჭდება, მაგრამ უნდა გვახსოვდეს რომ თუ ფუნქციას მასივს ვაწვდით პარამეტრად, უნდა გადავაწოდოთ ელემენტების რაოდენობაც. თავის დროზე C ენის ავტორმა დენის რიჩჩიმ აღნიშნა, პროგრამაზე პასუხისმგებელი ყოველთვის არის მხოლოდ პროგრამისტი.

როდესაც დინამიკურად შექმნილი მასივი აღარ არის საჭირო, მისთვის გამოყოფილ მეხსიერებას ვუბრუნებთ სისტემას შემდეგი მარტივი წინადადებებით:

```
delete[] p;
p = nullptr;
```

თავის დაზღვევის მიზნით, ხშირად გამართლებულია შემოწმდეს, მოხდა თუ არა შეკვეთილი მეხსიერების გამოყოფა. განხილულ მაგალითში, ეს გაკეთდებოდა ასე: სტრიქონის

```
int* mas = new int[20];
```

ნაცვლად ვწერთ:

```
int* mas = new (nothrow) int[20];
if (mas == nullptr) {
	cout << "Error assigning memory. Take measures" << endl;
}
```

შევნიშნოთ, რომ მასივი არის ობიექტების კოლექცია და მის წასაშლელად delete ოპერატორის და [] ოპერატორის ერთდროული გამოყენებაა საჭირო. ამისგან განსხვავებით, კლასის საშუალებით გაკეთებულ კონტეინერები ობიექტებია და მათთვის საკმარისია delete.

<<< დინამიკური მასივების მნიშვნელობების გადაცვლა - swap

ძალიან ხშირად, როდესაც განვიხილავთ რაიმე დინამიკურ პროცესს, მაგალითად განტოლებათა სისტემის ამოხსნას, ან დღეების მიხედვით ტემპერატურის სხვაობის შეფასებას, საჭირო ხდება ორი მასივისთვის მნიშვნელობების გადაცვლა. სტატიკური მასივებისთვის მნიშვნელობების გადაცვლა ნიშნავს, რომ მოგვიწევს წევრ-წევრად გადავცვალოთ მნიშვნელობები. ორ დინამიკურ მასივს შეგვიძლია სათაურები გადავუცვალოთ - პირველ მასივს გადავარქმევთ სახელს და დავარქმევთ მეორის სახელს, ხოლო მეორეს დავარქმევთ პირველი მასივის სახელს.

განვიხილოთ კონკრეტული მაგალითი.

მაგალითი 2. ვთქვათ, ფაილში "data.txt" წერია აქციათა ფასები დროის სხვადასხვა მომენტში, ცალ-ცალკე სტრიქონად (სინამდვილესთან უფრო მიახლოებული იქნება, თუ ამ ფასებს პერიოდულად შემოვიტანთ რაიმე სერვერიდან). ჩვენს ამოცანას წარმოადგენს ამ სტრიქონების

შედარება წევრ-წევრად, და თუ რომელიმე სტრიქონში, წინასთან შედარებით, ფასები დაეცემა აქციების 80% -ზე, შევწყვიტოთ პროცესი და დავბეჭდოთ საგანგაშო ინფორმაცია.

განხორციელება: თუ სტრიქონების რაოდენობა დიდია (რაც აუცილებლად ხდება რეალურ ამოცანებში), მაშინ ყველა სტრიქონის ერთდროული წაკითხვა ორგანზომილებიან კონტეინერში **არ ივარგებს** - დროის ყოველ მომენტში საკმარისია RAM -ში გვქონდეს შენახული ორი ერთმანეთის მომდევნო სტრიქონი, რომლებსაც ვადარებთ ერთმანეთს. თუ შედეგი საგანგაშოა, პროცესი წყდება და იბეჭდება გზავნილი, თუ არაა საგანგაშო, მაშინ ამ ორი სტრიქონიდან შედარებით ძველის ნაცვლად წავიკითხავთ მომდევნო სტრიქონს და პროცესი განმეორდება.

იმისათვის, რომ პროცესი მოვაქციოთ განმეორების შეტყობინებაში, შევთანხმდეთ რომ ორ მასივს, რომლებთანაც ვმუშაობთ, ვარქმევთ s0 და s1,- ამათგან ახალია s1. თუ პროცესი მეორდება, განხილულებიდან ახალი უნდა გახდეს მომდევნო ბიჯისთვის ძველი, ანუ s0, ხოლო მომდევნო სტრიქონი უნდა წავიკითხოთ s1-ში. როგორც ვხედავთ, საჭიროა ამ ორი მასივის მნიშვნელობების გადაცვლა. შემდეგ პროგრამაში, დინამიკური მასივების მნიშვნელობების გადაცვლა დაყვანილია მათი სახელების გადაცვლაზე:

```
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;
template<typename T>
void printMassive(T* name, int size)
{
	for (int i = 0; i<size; i++)
		cout << name[i] << "   ";
	cout << endl;
}
int main()
{
	ifstream ifs("data.txt");
	int n, m;                          //Number of shares and rows
	ifs >> n >> m;
	int counter(1), countChanges(0);
	int* s0 = new int[n];              //dynamic arrays for 2 stocks
	int* s1 = new int[n];
	for (int i = 0; i<n; i++)
		ifs >> s0[i];

	for (int i = 1; i<m; i++)
	{
		for (int j = 0; j<n; j++)
			ifs >> s1[j];
		for (int j = 0; j<n; j++)
			if (s1[j] < s0[j])
				countChanges++;
		if (countChanges > 0.7*n)
		{
			cout << "Worning! # Shares are collapsing!" << endl;
			printMassive(s0, n);
			printMassive(s1, n);
			cout << "counter = " << counter << endl;
			return 0;
		}
		int * p = s0;
		s0 = s1;
		s1 = p;
```

```
            counter++;
            countChanges = 0;
        }
        delete[] s0;
        delete[] s1;
        s0 = s1 = nullptr;
        cout << "No problems..." << endl;
}
```

ჩვენს ამოცანაში, მონაცემების ფაილს ჰქონდა სახე:

```
11  6
21  3   4   3   23  32  43  2   5   32 432
3   4   3   54  32  43  2   5   32 432  84
12  21  34  3   4   3   21  32  43  2   5
32 432  9   21  3   4   3   23  32  43  2
5   32  43  13   4   3   52  32  43  2   5
1   4   32  11   2   5   32 432  41  5   2
```

ხოლო პასუხად იბეჭდება:

```
Worning! # Shares are collapsing!
5   32   43   13   4   3   52   32   43   2   5
1   4   32   11   2   5   32   432   41   5   2
counter = 5
Press any key to continue . . .
```

<<< New და delete[] ოპერატორები ორგანზომილებიან მასივებთან.

ორ და მეტგანზომილებიან მასივებთან მუშაობა ბევრად რთულია. განსაკუთრებით რთულდება ფუნქციაში მათი პარამეტრად გადაცემა. საბედნიეროდ, ასეთი მასივების გამოყენებლობის ძირითადად სფეციფიკური ამოცანების გადაჭრისას ჩნდება.

განვიხილოთ მხოლოდ ორგანზომილებიანი დინამიკური მასივის შექმნა და მისი გაუქმებ, აგრეთვე, ასეთი მასივების გამოყენება რიცხვების მატრიცაში მაქსიმალური ელემენტის ინდექსის განსაზღვრისთვის.

C/C++ ენებში, რაიმე T ტიპის ელემენტებისგან შედგენილი ორგანზომილებიანი m მასივის ელემენტებზე წვდომა ხორციელდება ინდექსების გამოყენებით, მაგალითად: `m[i][j]`. ამასთან, აქ ჯერ გამოიყენება `i`, შემდეგ `j`. ასეთი ჩანაწერი არის მათემატიკურ აღნიშვნებთან უფრო მიახლოებული, რომლითაც „შეფუთულია“ პოინტერების არითმეტიკის აღნიშვნები. კერძოდ, `m[i][j]` იგივეა რაც `*(m[i]+j)`, რაც ნიშნავს რომ `m[i]` არის პოინტერი `*T` ტიპზე. მაგრამ, ანალოგიური მოსაზრებების გამო, ეს ნიშნავს, რომ `m` არის პოინტერი `**T` ტიპზე.

ამგვარად, თუ გვინდა გავაკეთოთ განაცხადი რაიმე T ტიპის ელემენტებისგან შედგენილ დინამიკურ ორგანზომილებიან m მასივზე, უნდა დავწეროთ:

```
T** m = new T*[n];
```

სადაც n ელემენტების რაოდენობაა. მაგალითად

```
int** p = new int*[20];
```

ამის შემდეგ უკვე საჭიროა ვიზრუნოთ, რომ შექმნილი მასივის ყოველმა ელემენტმა ისევ `new` ოპერატორის საშუალებით მნიშვნელობად მიიღოს დაჯავშნილი მეხსიერების მონაკვეთის დასაწყისი. უკვე ამის შემდეგ შეგვიძლია შექმნილი მასივის ელემენტებთან ინდექსების საშუალებით მუშაობა.

როდესაც შექმნილი მასივი ამოწურავს თავის ფუნქციას, უნდა გავათავისუფლოთ ყველა ერთგანზომილებიანი მასივის მეხსიერება, რაც ჩვენ მოვითხოვეთ. უნდა გავათავისუფლოთ

ორგანზომილებიანი მასივისთვის გამოყოფილი მეხსიერებაც. ბოლოს, პოინტერი, რომლიც შეესაბამებოდა ორგანზომილებიან მასივს, „არ უნდა გაველურდეს“, ანუ m = nullptr;

ნიმუშად განვიხილოთ ასეთი მაგალითი.

მაგალითი 3. ვთქვათ, ფაილში „matrix.txt“ პირველ სტრიქონად წერია ორი მთელი რიცხვი (სვეტების და სტრიქონების რაოდენობები), ხოლო შემდეგ წერია ნამდვილი რიცხვების ორგანზომილებიანი ცხრილი. ჩვენს ამოცანას წარმოადგენს ამ ცხრილის გადაწერა ორგანზომილებიან დინამიკურ მასივში ანუ მატრიცაში, შესაბამისი ფუნქციის გამოყენებით (რომელიც ასევე ჩვენ უნდა შევქმნათ) მატრიცაში უდიდესი ელემენტის სტრიქონისა და სვეტის მოძებნა. მაქსიმალური ელემენტის კოორდინატების მოძებნის შემდეგ უნდა გავაუქმოთ ჩვენს მიერ შექმნილი ორგანზომილებიანი მასივი.

განხორციელება: დავუშვათ, მონაცემების ფაილში წერია:

```
11  5
2.1  3.   4.   3.   2.3  32.  4.3  2.   .5   3.2 43.2
.3   4.   3.   5.4  3.2  43.  2.   5.   32.  4.32  84.
12.  21.  34.  3.   4.   3.   2.1  3.2  43.  2.   5.
32.  432.  9.   .21  .3   4.   3.   23.  3.2  43.  2.
5.   3.2  43.  13.   4.   .3   52.  3.2  43.  2.   5.  2
```

C++ ენაში გლობალური ცვლადების გამოყენება არაა მიზანშეწონილი, რადგან სხვა, უფრო დახვეწილი ტექნიკა არსებობს (სახელთა სივრცე, რასაც ბოლო ლექციებზე განვიხილავთ). თუმცა ეს საკითხები, რასაც ახლა ვარჩევთ, უფრო C ენის შემადგენელია. ამიტომ გავიმარტივოთ საქმე და სტრიქონებისა და სვეტების რაოდენობაზე განაცხადი გავაკეთოთ გლობალურად, main() ფუნქციის წინ, რომ მათი დანახვა შეეძლოთ გამოყენებულ ფუნქციებს.

საბოლოოდ, ჩვენს პროგრამას ექნება საკმაოდ მარტივი სახე:

```
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;

template<typename T>
int maximal(T* name, int size)
{
	int index(0);
	for (int i = 1; i<size; i++)
		if (name[index] < name[i])
			index = i;
	return index;
}
//Number of columns and rows
int n, m;
template<typename T, typename BinaryPredicate>
int maximal(T* name, int size, BinaryPredicate p)
{
	int index(0);
	for (int i = 1; i<size; i++)
		if (p(name[index], name[i]))
			index = i;
	return index;
}
bool p(double *a, double *b)
{
	int i = maximal(a, n);
	int j = maximal(b, n);
```

```
	return (a[i] < b[j]);
}
int main()
{
	ifstream ifs("data.txt");
	ifs >> n >> m;
	double** mas = new double*[m];
	for (int i = 0; i<m; i++)
		mas[i] = new double[n];

	for (int i = 0; i< m; i++)
		for (int j = 0; j< n; j++)
			ifs >> mas[i][j];

	int row = maximal(mas, m, p);
	int column = maximal(mas[row], n);

	cout << "row: " << row << endl;
	cout << "column: " << column << endl;
	//free
	for (int i = 0; i<m; i++)
		delete[] mas[i];
	delete[] mas;
	mas = nullptr;
}
```

ყურადღება მივაქციოთ, თუ როგორ ხდება მაქსიმუმის კოორდინატების განსაზღვრა. ჯერ ვეძებთ რომელ სტრიქონშია მაქსიმალური. შემდეგ მოძებნილ სტრიქონში ვეძებთ მაქსიმალური ელემენტის შესაბამის სვეტს. შეიძლება ვიფიქროთ, რომ ასე ერთი სტრიქონისთვის ზედმეტად ვიანგარიშეთ მაქსიმალური ელემენტი, მაგრამ სამაგიეროდ ყოველ ნაბიჯზე არ ვიმახსოვრებდით ხელახლა დამატებით ცვლადებს.

<<< სავარჯიშოები

1. დინამიკურად შექმენით ნამდვილი რიცხვების 6 ელემენტიანი მასივი. მასივის ელემენტებში მნიშვნელობები შეიყვანეთ კლავიატურიდან. შექმენით მასივის ელემენტების ბეჭდვის ფუნქციის თარგი და მისი გამოყენებით დაბეჭდეთ ეს მასივი. შემდეგ შეიტანეთ ცვლილება მასივში და ისევ დაბეჭდეთ.
2. დინამიკურად შექმენით სიმბოლოების 5 ელემენტიანი მასივი. მასივის ელემენტებში მნიშვნელობები შეიყვანეთ კლავიატურიდან. იპოვეთ და დაბეჭდეთ უმცირესი სიმბოლო. გაოიყენეთ ბიბლიოთეკის ფუნქცია.
3. იგივე ამოცანები გააკეთეთ დინამიკურად სიის, ვექტორის და დეკის გამოყენებით.
4. შექმენით კლასები ზღვისთვის, ქალაქისთვის და მთისთვის. გარდა სახელისა, მათი ველები, შესაბამისად არის ფართობი, მოსახლეობის რაოდენობა და სიმაღლე.

   მთავარ პროგრამაში დინამიკურად გააკეთეთ რამდენიმე ობიექტი სხვადასხვა კონსტრუქტორის გამოყენებით.
5. შექმენით ფუნქცია, რომელიც დაბეჭდავს სხვადასხვა ტიპის მასივის ელემენტებს რამდენიმე სვეტად. კონტეინერი და სვეტების რაოდენობა ფუნქციის პარამეტრებია. მთავარ პროგრამაში დინამიკურად შექმენით სხვადასხვა ტიპის რამდენიმე მასივი და დაბეჭდეთ ისინი. როდესაც მასივი საჭირო არაა, გააუქმეთ იგი.