Tema 06. Dinaminė atmintis.

Turinys

[Dinaminės atminties valdymo funkcijos 2](#_Toc371600807)

[Statiniai ir dinaminiai (daugiamačiai) masyvai 5](#_Toc371600808)

[Pavyzdys: Dvimatis masyvas dinaminėje atmintyje 7](#_Toc371600809)

Dinaminės atminties valdymo funkcijos

1. C neturi vidinių kalbos priemonių darbui su dinamine atmintimi. Manipuliacijoms su dinamine atmintimi naudojamos standartinės bibliotekos funkcijos, aprašytos <stdlib.h> ir/arba <alloc.h>.
   1. Dvi dinaminės atminties paskirstymo

**void \*** malloc**(unsigned** size**);**

**void \*** calloc**(unsigned** num **, unsigned** size**);**

ir viena perpaskirstymo

**void \*** realloc**(void \*** ptr **, unsigned** size**);**

funkcijos grąžina dinaminėje atmintyje paskirtos srities adresą. Šis adresas yra netipizuotas.   
Funkcijoms malloc() ir realloc() reikalingos atminties dydis baitais nusakomas parametre size.   
Funkcijai calloc()parametruose size ir num, todėl paskirtos atminties dydis bus lygus parametrų sandaugai. Be to, ši funkcija apnulina paskirtą atmintį.   
Funkcija realloc()vietoj parametre ptr adresu nurodytos atminties paskiria naują (tik tuo atveju, jei reikia!), perkopijuoja senosios turinį į naująją ir atlaisvina senąją.

Visos trys funkcijos grąžina NULL , jei paskirti atminties nepavyko.

* 1. Paskirta atmintis atlaisvinama funkcija (procedūra)

**void** free**(void \*** ptr**);**

* 1. Naudingos manipuliavimo su dinamine atmintimi funkcijos yra aprašytos <alloc.h>, pvz.:

**void \*** memcpy**(void \*** dest **, void \*** src **, unsigned** count**);**

**void \*** memset**(void \*** dest **, int** c **, unsigned** count**);**

Svarbu! Adresai dest ir src turi būti dinaminės atminties (jokiais būdais ne statinės ar automatinės) adresais.

* 1. Pavyzdys:

**int \***ptr**;**

ptr **= /\*(int\*)\*/** malloc**(sizeof(int));**

free**(**ptr**);**

* 1. Pavyzdys:

**int** dim**;**

**char \***memptr**;**

printf**(”**Kiek atminties reikia eilutei?\n**”);**

scanf**(”**%d**”,&**dim**);**

**if (**NULL **== (**memptr **= (char\*)**calloc**(**dim**,sizeof(char)))) {**

printf**(”**Neliko laisvos atminties\n**”);**

exit**(**1**);**

**}**

...

free**(**memptr**);**

1. C++ turi dinaminės atminties paskirstymo ir atlaisvinimo operacijas: **new** ir **delete**.
   1. Kreipinio į dinaminės atminties paskyrimo operaciją sintaksė  **new** *tipas***;** , pvz.:

**int \***ptr**;**

ptr **= new int;**

Lyginant su C funkcijomis, galima pastebėti 3 new operacijos privalumus: pirma, nereikia vargintis dėl reikalingos atminties dydžio nurodymo – jį netiesiogiai nusako operandas *tipas*; antra, operacija grąžina tipizuotą rodyklę – jos tipas yra rodyklė į *tipas*; trečia, rodyklės galima inicijuoti dinaminės atminties adresais jų apibrėžimo metu, pvz.:

**int \***ptr **= new int;**

* 1. Dinaminę atmintį galima paskirti ir masyvui, pvz.:

cin >> dim**;**

memptr **= new char[**dim**];**

* 1. Paskirtos dinaminės atminties atlaisvinimo sintaksė įprastam tipo duomeniui **delete** *adresas***;** ir **delete** **[]** *adresas***;** masyvui, pvz.:

**delete** ptr**; delete** **[]** memptr**;**

1. Skirtingai nei kitose kalbose (aplinkose), C ir C++ sistemos neatlieka automatinio nebenaudotinos dinaminės atminties atlaisvinimo (ši kitų kalbų (sistemų) savybė angliškai vadinama ***garbage collection***). Todėl programuotojas turi pats pasirūpinti ne tik dinaminės atminties paskyrimu, bet ir deramu ir savalaikiu jos atlaisvinimu.

Statiniai ir dinaminiai (daugiamačiai) masyvai

Taisyklė, kad masyvai realizuojami rodyklėmis nėra taikoma rekursyviai, nes masyvų masyvas (2-matis masyvas),pvz.

**int** a2i**[**ROWS**][**COLS**];**

sistemos yra „suprantamas“ **ne** kaip rodyklė į rodyklę:

**int\*\*** ppi**;**

o kaip **rodyklė į** COLS dydžio int **masyvą**:

**int (\***pai**)[**COLS**];**

Tad, prisimenant vienmačio masyvo „sprendimą“ naudojant rodyklių aritmetiką, į tesingą pretenduojantis priskyrimas

***ppi=a2i*;**

nors ir leistinas (perspėjimas), tačiau yra klaidingas. Teisingas yra priskyrimas

**pai=a2i;**

Išspręskime kreipinį (t.y. reiškinį) į 2-mačio masyvo elementą **a2i[i][j]** rodyklių aritmetikos pagalba.

Kaip ir 1-mačio masyvo atveju, reiškinių **a2i** ir **a2i+i** tipas yra tas pats, tik 2-mačio masyvo atveju tai - rodyklė į COLS dydžio int masyvą (1-mačio atveju - rodyklė į int), o reikšmė lygi i-jo COLS dydžio masyvo pradžios adresui!

Tuomet, **\*(a2i+i)** reiškinio, kuris ir yra reiškinio **a2i[i]** sprendinys, tipas turėtų būti „masyvas“, t.y. rodyklė į int, o reikšmė lygi i-jo COLS dydžio masyvo 0‑nio elemento adresui, t.y. tam pačiam, tačiau kito tipo adresui!

Taigi, dabar turime įprastą :) 1-matį COLS dydžio masyvą **\*(a2i+i)[COLS]** , o kreipinio į j-jį jo elementą **\*(a2i+i)[j]** sprendinys (2-ai dimensijai) yra **\*(\*(a2i+i)+j)** .

Reziumuokime:

**a2i[i][j] == \*(a2i+i)[j]== \*(\*(a2i+i)+j) == \*(\*(pai+i)+j) == \*(pai+i)[j] == pai[i][j]**

Tema išties sudėtinga, todėl, didžiąja dalimi, paliekama savarankiškoms studijoms (žr. programų pavyzdžius).

Taip pat paminėtina, kad

**int \***api**[**COLS**];**

apibrėžia COLS dydžio masyvą, sudarytą iš rodyklių į int.

(plačiau apie aprašų skaitymo „algoritmą“ čia: [www.codeproject.com](http://www.codeproject.com/Articles/7042/How-to-interpret-complex-C-C-declarations)).

Dvimatį masyvą apdorojančios funkcijos parametras turėtų būti arba metodiška atvirojo indeksavimo forma

f**(int** ai**[][**COLS**]) {...}**

arba rodyklių forma

f**(int (\***pai**)[**COLS**]) {...}**

Reikia atkreipti dėmesį, kad tik antrosios dimensijos (bendru atveju, visų išskyrus pirmąją) dydis yra fiksuotas. T.y. šią funkciją galima iškviesti ne tik ai**[**ROWS**][**COLS**]** bet ir pvz. ai**[**ROWS+1**][**COLS**]**, tačiau negalima ai**[**ROWS**][**COLS+1**].**

Žymiai lankstesnis ir efektyvesnis būdas yra „valdyti“ daugiamačius masyvus dinaminėje atmintyje (žr. pavyzdį).

Pavyzdys: Dvimatis masyvas dinaminėje atmintyje

#include <stdio.h>

#include <stdlib.h>

void init\_array (long\*\*, int, int);

void print\_array(long\*\*, int, int);

int main() {

**long\*\* matrix**;

int rows, cols;

int i;

scanf("%d%d", &rows, &cols);

**matrix** = (long\*\*)malloc(sizeof(long\*)\*rows);

for (i=0 ; i < rows ; i++) {

**matrix[i]** = (long\*)malloc(sizeof(long)\*cols);

}

// alloc\_array (long\*\*\* matrix) { \*matrix = (long\*\*)malloc(sizeof(long\*)\*rows);...}

init\_array (matrix, rows, cols);

print\_array(matrix, rows, cols);

for (i=0 ; i < rows ; i++) {

free(matrix[i]);

}

free(matrix);

// delete\_array (long\*\*) {...}

return 0;

}

void init\_array(long\*\* matrix, int rows,int cols) {

int i,j;

for (i=0 ; i < rows ; i++)

for (j=0 ; j < cols ; j++)

**matrix[i][j]** = i+j;

/\* ***\*(\*(matrix+i)+j)*** = i+j; \*/

}

void print\_array(long\*\* matrix, int rows,int cols) {

int i,j;

for (i=0 ; i < rows ; i++) {

for (j=0 ; j < cols ; j++)

printf ("%ld\t", **matrix[i][j]**);

printf ("\n");

}

}