Dynamiczna alokacja tablic wielowymiarowych

Tydzień temu nauczyliśmy się dynamicznej alokacji pamięci dla tablic jednowymiarowych. Dynamiczna oznacza tyle, że rozmiar tablicy, jaką chcemy zaalokować, znamy dopiero w momencie wykonywania programu, tzn., że nie jesteśmy w stanie go określić statycznie (daną, konkretną liczbą) na etapie kompilacji. Kod do dynamicznej alokacji tablic jednowymiarowych wyglądał tak:

```
#include <stdlib.h>

void main()
{
   double *tab;
   int n;
   printf("Podaj n:\n");
   scanf("%d, &n);

   tab = (double*)malloc(n*sizeof(double));
   // Tu mozesz wykonywac operacje na tablicy
   free(tab);
}
```

Dziś nauczymy się dwóch nowych rzeczy: stosowania w języku C tablic wielowymiarowych (alokowanych statycznie) oraz ich dynamicznej alokacji.

Tablice wielowymiarowe

Język C pozwala na stosowanie tablic wielowymiarowych. Do tej pory przez kilka tygodni używaliśmy jedynie tablic jednowymiarowych. Potocznie często określaliśmy je *wektorami*. Wyobraźmy sobie - tablica dwuwymiarowa doskonale nadaje się np. do przechowywania macierzy.[1] Przyjrzyjmy się więc fragmentowi kodu, który zadeklaruje dwuwymiarową tablicę o wymiarze 3x4. Możemy to utożsamić z reprezentacją macierzy o takim samym wymiarze.

```
void main()
{
   double A[3][4];

   // Tu mozemy przypisac wartosci kolejnym elementom:
   A[0][0] = 1;
   A[0][1] = 1.5;
   A[0][2] = 0;
   A[0][3] = -2.7;
   ...
   A[2][3] = 8;
   // Tu mamy wypelniona macierz
   // Mozemy wykonywac obliczenia
}
```

Zauważmy, że w przypadku tablic deklarowanych statycznie nie ma potrzeby ich zwalniania. Kompilator sam o to dba (jak w przypadku wszystkich zmiennych, które do tej pory deklarowaliśmy - one też są automatycznie niszczone przez kompilator). Powyższą macierz możemy też wypełnić wartościami w nieco zgrabniejszy sposób niż przez wypisanie kolejnych dwunastu linijek przypisań. Możemy to zrobić na liście inicjalizacyjnej od razu na etapie deklaracji tablicy. Dokonuje się tego tak:

```
double A[3][4] = {{1, 1.5, 0, -2.7}, {-3, 2.5, 7, 0}, {0, 1, -3, 8}};
```

W ten sposób stworzymy poniższą macierz:

$$\mathbf{A} = \begin{pmatrix} 1 & 1.5 & 0 & -2.7 \\ -3 & 2.5 & 7 & 0 \\ 0 & 1 & -3 & 8 \end{pmatrix}$$

Pozostaje wytłumaczyć jeszcze, w jaki sposób odwołujemy się do elementów w dwuwymiarowej tablicy. Robimy to analogicznie do tablicy jednowymiarowej, tylko tym razem musimy podać dwa indeksy. Tak więc do elementu macierzy a_{32}

[1] Tak naprawdę wiele więcej struktur - często nawet zupełnie niematematycznych możemy trzymać w dwuwymiarowych tablicach - np. programując grę w szachy moglibyśmy użyć dwuwymiarowej tablicy o wymiarze 8 x 8 i w odpowiednie pola tej tablicy wpisywać liczby, które symbolizowałyby konkretne figurę. A gra w statki? Można podobnie. Z drugiej strony w praktycznych obliczeniach numerycznych wielkie macierze o rozmiarze rzędu kilkuset tysięcy do kilku milionów i większe przechowuje się w postaci wektora. W praktycznych zagadnieniach są to niemal zawsze macierze rzadkie, tzn. takie, które w stosunku do całkowitej liczby swoich elementów mają bardzo niewiele elementów, które nie są zerem. Taką macierz łatwo jest trzymać w pamięci jako wektor (przyjmując specjalny format, który pomija wszystkie zera - np. tzw. format CSR *(ang. compressed sparse row)* jest szeroko stosowanym formatem zapisania macierzy rzadkiej w trzech wektorach). Tak wielka macierz przechowywana jawnie najpewniej nie zmieściłaby się w pamięci żadnego dostępnego nam komputera.

odwołamy się przez napisanie `a[2][1]`. W ten sposób możemy wyłuskać wartość przechowywaną pod tym elementem lub operatorem = przypisać temu elementowi nową wartość.

Ćwiczenia

W funkcji `main` napisz fragment kodu, w którym zadeklarujesz i zainicjalizujesz dowolnymi wartościami dwie różne tablice dwuwymiarowe. Jedna ma przechowywać macierz kwadratową o wymiarze 2, a druga z nich macierz kwadratową o wymiarze 3. Napisz kod, który dla każdej z tych macierzy policzy wyznacznik.

Dynamiczna alokacja

Czas jednak na dynamiczną alokację. Dwuwymiarową tablicę o rozmiarze M x N zaalokujemy w następujący sposób: stworzymy tablicę jednowymiarową o rozmiarze M (umówmy się, że ona będzie wskazywać na początek każdego z wierszy), po czym każdemu z elementów tej tablicy zaalokujemy blok o długości N (to będą jednowymiarowe tablice do przechowywania kolejnych wierszy). De facto będziemy mieli w pamięci M bloków, każdy długości N. Spójrzmy na kod.

```
double **A;
A = (double**)malloc(M*sizeof(double*));
```

Zauważmy, że tydzień temu alokowaliśmy jednowymiarową tablicę jako wskaźnik. Tym razem będziemy mieć tablicę dwuwymiarową, więc używamy podwójnego wskaźnika. Dlatego w instrukcji powyżej blok pamięci zwracany przez funkcję `malloc` rzutujemy na podwójny wskaźnik `double**`. Musimy też obliczyć, ile miejsca potrzebujemy. Przechowywać będziemy wskaźniki (do odpowiednich tablic jednowymiarowych przechowujących wiersze), dlatego jako argument funkcji `sizeof` podajemy `double*`. Teraz alokujemy tablice jednowymiarowe do przechowywania wierszy.

```
for(int i = 0; i<M; ++i)
   A[i] = (double*)malloc(N*sizeof(double));
```

Powyżej każdemu z elementów pierwszej tablicy przypisaliśmy tablicę do przechowywania każdego z wierszy. Tym razem blok zwrócony przez funkcję `malloc` rzutujemy na typ `double*`, a argumentem funkcji `sizeof` jest typ zmiennej przechowywanej w tej tablicy, czyli już zwykła zmienna `double`, a nie wskaźnik do niej. Tablica dwuwymiarowa jest już gotowa - możemy jej używać.

Po zakończeniu pracy z tablicą, trzeba koniecznie zwolnić pamięć przez nią wykorzystywaną. Teraz trzeba operacje wykonać od końca! Tzn. najpierw zwalniamy każdy z wierszy, a na końcu zwolnimy pierwotną tablicę wskaźników do wierszy. Dokonuje tego poniższy kod.

```
for(int i = 0; i<M; ++i)
   free(A[i]);

free(A);
```

Podsumowanie

Zbierzmy wszystkie instrukcje w jednym miejscu. Chcemy zaalokować dwuwymiarową tablicę zmiennych typu `int`. Dokonujemy tego tak:

```
// Alokacja pamieci
int **A;
A = (int**)malloc(M*sizeof(int*));
for(int i = 0; i<M; ++i)
   A[i] = (int*)malloc(N*sizeof(int));

// Tu mozemy wykonywac dowolne operacje na tablicy

// Zwolnienie pamieci
for(int i = 0; i<M; ++i)
   free(A[i]);
free(A);
```

Ćwiczenia

- Napisz funkcję, która jako argumenty przyjmie podwójny wskaźnik (wskaźnik do dynamicznie alokowanej tablicy dwuwymiarowej) oraz liczbę kolumn i wierszy macierzy, po czym dokona wydruku macierzy na ekran w naturalnej postaci, do jakiej jesteśmy przyzwyczajeni dla macierzy.

- Napisz funkcję `maxAbsAij`, która dla danej macierzy zwróci do funkcji `main` największy co do modułu element tej macierzy oraz jego indeksy i, j.
- Napisz funkcję `minAbsAij`, która dla danej macierzy zwróci do funkcji `main` najmniejszy co do modułu element tej macierzy oraz jego indeksy i, j.
- W funkcji main zaalokuj w sposób dynamiczny miejsce dla macierzy 3x3 oraz dwóch wektorów trzyelementowych. Wypełnij macierz i jeden z wektorów dowolnymi wartościami, po czym napisz w funkcji `main` kod, który dokona przemnożenia danej macierzy przez dany wektor i wynik mnożenia wpisze do drugiego z wektorów. Mnożenie macierzy przez wektor określone jest wzorem

$$w_i = \sum_{j=0}^{n-1} a_{ij} v_j$$

.
- Zamknij powyższe operacje w funkcji o nagłówku `c++void MatVecMultiply(doub **A, double *v1, double *v2, int n)` i dokonaj wywołania z funkcji `main`.
- Zmodyfikuj powyższy program tak, żeby rozmiar n był wczytywany z klawiatury, elementy tablicy były generowane zgodnie ze wzorem $a_{ij} = \frac{i+1}{j+1}, (i, j = 0, ..., n-1)$, zaś elementy wektora wg wzoru $v_i = i + 1, (i = 0, ..., n-1)$. Obliczaj iloczyn takiej macierzy przez ten wektor, korzystając ze swojej funkcji `MatVecMultiply`. Wynik wyświetlaj na ekranie oraz sprawdź, czy otrzymujesz poprawny wynik.[2]
- Napisz funkcję `MatMatMultiply(double **A, double **B, double **C, int mA, int nA, int mB, int nB)` służącą do mnożenia dwóch macierzy prostokątnych (**A** o wymarze $m_A \times n_A$ i **B** o wymiarze $m_B \times n_B$) i wpisującą wynik do macierzy **C** (zadbaj w funkcji `main` o to, aby pamięć zaalokowana dla macierzy **C** była odpowiedniej wielkości - zgodnej z regułami mnożenia macierzy). **Uwaga:** Pamiętaj, aby koniecznie zwolnić wszelką dynamicznie alokowaną pamięć. [3]

Alokacja wewnątrz funkcji

Dlaczego poniższy kod nie działa poprawnie?

[2] Dla takiej macierzy i takiego wektora bardzo łatwo jest wygenerować analityczny wynik. Wypisz sobie małą macierz wg zadanego wzoru i odpowiadający wektor i na pewno szybko zauważysz prawidłowość. Będziesz wiedzieć, jaki wynik powinien dać program. Tak się testuje programy na wczesnych etapach rozwoju.

[3] Niezwolnienie pamięci prowadzi do jej wycieków i w przypadku pewnych operacji wykonywanych w pętlach może doprowadzić do tego, że Twój program wykorzysta całą pamięć operacyjną komputera i przestanie działać.

```
void AllocateAndFillArrayBAD(int **A, int M, int N)
{
    A = (int**)malloc(M * sizeof(int*));
    for (int i = 0; i<M; ++i)
        A[i] = (int*)malloc(N * sizeof(int));


    // Tu mozemy wykonywac dowolne operacje na tablicy
    int count = 0;
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            A[i][j] = ++count; // OR *(*(A+i)+j) = ++count

    //drukuj
    for (int i = 0; i < M; ++i)
    {
        printf_s("\n");
        for (int j = 0; j < N; ++j)
        {
            printf_s(" %d ", A[i][j]);
        }
    }

    printf_s("\n");
}

int main()
{
   int M = 4;
    int N = 3;
    int **A = nullptr;

   AllocateAndFillArrayBAD(A, M, N);

   //gdzie są dane?
    for (int i = 0; i < M; ++i)
    {
        printf_s("\n");
        for (int j = 0; j < N; ++j)
```

```
        {
            printf_s(" %d ", A[i][j]);
        }
    }


    // Zwolnienie pamieci
    for (int i = 0; i<M; ++i)
        free(A[i]);

    free(A);
}
```

Popraw deklarację i ciało funkcji `AllocateAndFillArrayBAD`. - Wskazówka: prawidłowa deklaracja w języku C powinna wyglądać tak:
`void AllocateAndFillArray(int ***A, int M, int N)`

Dla ambitnych *

Zastanów się, jak wyglądałaby dynamiczna alokacja i zwolnienie pamięci dla tablicy trójwymiarowej. Na przykład, gdybyś chciał napisać grę w trójwymiarowe kółko i krzyżyk. Skonsultuj z prowadzącym kod dla takiego przypadku.