Tydzień temu nauczyliśmy się jak dynamicznie alokować pamięć dla tablicy jednowymiarowej. Wyraz „dynamicznie" oznaczał tyle, że rozmiar tablicy, którą chcemy zaalokować, poznamy dopiero w trakcie wykonywania programu i nie jesteśmy w stanie go określić (konkretną liczbą) na etapie kompilacji. Kod dokonujący dynamicznej alokacji tablicy jednowymiarowej wygląda tak:

```
#include <stdlib.h>

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

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

Dziś nauczymy się:

- stosowania statycznych tablic wielowymiarowych oraz
- dynamicznej alokacji tablic wielowymiarowych.

Statyczne tablice wielowymiarowe

Język C pozwala na używanie tablic wielowymiarowych. Wyobraźmy sobie – tablica dwuwymiarowa doskonale nadaje się do przechowywania np. macierzy[1].

[1] Tak naprawdę, tablica może przechowywać o wiele więcej struktur. Przykładowo programując grę w szachy moglibyśmy użyć dwuwymiarowej tablicy o wymiarze $8x8$ i w odpowiednie pola tej tablicy wpisać liczby, które symbolizowałyby konkretną figurę. A gra w statki? Można podobnie. Z drugiej strony, w obliczeniach numerycznych wielkie macierze o rozmiarze rzędu kilkuset tysięcy do kilku milionów elementów i większe, przechowuje się w postaci tablicy jednowymiarowej. W zagadnieniach inżynierskich macierze są często rzadkie, tzn. takie, które w stosunku do całkowitej liczby swoich elementów mają bardzo niewiele elementów, które nie są zerem. Taką macierz efektywnie jest trzymać w pamięci jako tablicę jednowymiarową (przyjmując specjalny format, który pomija wszystkie zera – np. format CSR *(ang. Compressed Sparse Row)* jest szeroko stosowanym formatem zapisywania macierzy rzadkiej w trzech tablicach jednoelementowych). Tak wielka macierz przechowywana jawnie najpewniej nie zmieściłaby się w pamięci żadnego dostępnego nam komputera.

Przyjrzyjmy się więc fragmentowi kodu, który zadeklaruje dwuwymiarową tablicę o wymiarze 3×4. Możemy ją utożsamić z macierzą o takim samym wymiarze.

```
void main() {
   double A[3][4];        // deklaracja tablicy 2-wymiarowej

   A[0][0] = 1.;          // przypisanie wartosci
   A[0][1] = 1.5;         // poszczegolnym elementom
   A[0][2] = 0.;
   A[0][3] = -2.7;
   //...
   A[2][3] = 8.;
}
```

Przypomnijmy, że w przypadku tablic deklarowanych statycznie nie ma potrzeby ich zwalniania. Kompilator sam o to dba (tak 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 zapisanie dwunastu kolejnych linijek przypisań. Możemy to zrobić już w trakcie deklaracji tablicy, dzięki liście inicjalizującej. Dokonuje się tego tak (dla przypomnienia: znak „\" oznacza, że długa instrukcja będzie kontynuowana w następnej linii):

```
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 podajemy dwa indeksy. Wartość pierwszego indeksu to numer wiersza, drugiego – numer kolumny. Przykładowo, do elementu macierzy a_{32} odwołamy się przez napisanie `a[2][1]`. W ten sposób możemy wyłuskać wartość przechowywaną pod tym elementem lub za pomocą operatora „=" przypisać temu elementowi nową wartość.

Ćwiczenia

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

Dynamiczna alokacja tablic wielowymiarowych

Czas na dynamiczną alokację. Dwuwymiarową tablicę o rozmiarze $N \times M$ alokujemy w następujący sposób:

- tworzymy jednowymiarową tablicę wskaźników o rozmiarze N (każdy jej element będzie wskazywać na początek odpowiadającego mu wiersza),
- dla każdego z elementów tej tablicy alokujemy blok o długości M (będą to jednowymiarowe tablice do przechowywania wierszy).

Podsumowując, będziemy mieli w pamięci N bloków, każdy o długości M. Spójrzmy na poniższy kod:

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

Przypomnijmy sobie, że jednowymiarową tablicę alokowaliśmy wykorzystując wskaźnik do typu `double`. Tym razem będziemy alokować wskaźnik do wskaźnika na typ `double`. Dlatego używamy „podwójnego" wskaźnika. W instrukcji powyżej, adres zwracany przez funkcję `malloc()` rzutujemy więc na podwójny wskaźnik `double **`. Wywołując funkcję `malloc()` musimy wiedzieć, ile miejsca potrzebujemy. Przechowywać będziemy wskaźniki (do odpowiednich tablic jednowymiarowych przechowujących wiersze), dlatego jako argument funkcji `sizeof()` podajemy `double *`.

Teraz możemy zaalokować tablice jednowymiarowe do przechowywania wierszy:

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

Każdemu z elementów pierwszej tablicy przyporządkowaliśmy tablicę do przechowywania każdego z wierszy. Tym razem adres zwrócony przez funkcję `malloc()` rzutujemy na typ `double *`, a argumentem funkcji `sizeof()` jest typ zmiennej przechowywanej w tej tablicy, czyli już zwykła zmienna `double`. Tablica dwuwymiarowa jest już gotowa – możemy jej używać.

Po zakończeniu pracy z tablicą, trzeba koniecznie **zwolnić pamięć** wykorzystywaną przez nią. Najpierw zwalniamy tablice odpowiadające każdemu z wierszy, a na końcu zwalniamy pierwotną tablicę wskaźników do wierszy. Dokonuje tego poniższy kod:

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

/* wykonujemy dowolne operacje na tablicy */

/* zwalniamy pamiec */
for(int i = 0; i < N; ++i)
  free(A[i]);
free(A);
```

Ćwiczenia

1. Zaalokuj w funkcji `main` w sposób dynamiczny miejsce dla macierzy A o wymiarze $[3 \times 3]$ oraz dwóch 3-elementowych wektorów $v1$ i $v2$. Wypełnij macierz i jeden z wektorów dowolnymi liczbami.
2. Napisz funkcję `drukujMacierz`, która:

- jako argumenty przyjmuje: podwójny wskaźnik (wskaźnik do dynamicznie alokowanej tablicy dwuwymiarowej) oraz liczbę kolumn i liczbę wierszy macierzy,
- dokonuje wydruku macierzy na ekran w postaci do jakiej jesteśmy przyzwyczajeni z lekcji algebry.

3. Napisz funkcję `maxElem`, która dla zadanej macierzy zwraca do funkcji `main` wartość największego co do modułu elementu tej macierzy oraz jego indeksy i i j.
4. Napisz w funkcji `main` kod, który dokona mnożenia macierzy A przez wektor $v1$ a wynik zapisze do wektora $v2$. Mnożenie macierzy przez wektor określone jest wzorem:
$$w_i = \sum_{j=0}^{m-1} a_{ij} v_j$$
5. Zamknij powyższe operacje w funkcji o nagłówku

```
void matVecMultiply(double **A, double *v1, double *v2, int n)
```

i dokonaj wywołania z funkcji `main`. 6. Zmodyfikuj powyższy program tak, aby rozmiar n był wczytywany z klawiatury. Zaś elementy tablicy były generowane zgodnie ze wzorem
$$a_{ij} = \frac{i+1}{j+1}, (i, j = 0, \ldots, n-1)$$
a elementy wektora według wzoru
$$v_i = i + 1, (i = 0, \ldots, n-1)$$
Następnie oblicz iloczyn tej macierzy przez ten wektor, korzystając ze swojej funkcji `matVecMultiply`. Wynik wyświetl na ekranie oraz sprawdź czy jest poprawny[2]. 7. Napisz funkcję

```
matMatMultiply(double **A, double **B, double **C,\
               int nA, int mA, int nB, int mB)
```

służącą do mnożenia dwóch macierzy prostokątnych (**A** o wymiarze $n_A \times m_A$ i **B** o wymiarze $n_B \times m_B$). Wynik powinien być zapisany 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 zwolnić dynamicznie alokowaną pamięć[3].

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

Alokacja pamięci wewnątrz funkcji

Zastanów się dlaczego poniższy kod nie działa poprawnie?

```
void initialize_vec_BAD(int *ptr, int N) {
  ptr = (int *)malloc(N * sizeof(int));
  for (int j = 0; j < N; ++j)
    ptr[j] = 3 * j;

  printf("Wewn. funkcji initialize:\n");
  for (int j = 0; j < N; ++j) {
    printf("%d ", ptr[j]);
  }
  printf("\n");
}

int main() {
  int N = 10;
  int *v;

  initialize_vec_BAD(v, N);

  printf("Wewn. funkcji main:\n");
  for (int j = 0; j < N; ++j){
    printf("%d ", v[j]);
  }
  printf("\n");

  free(v);

  getchar();
}
```

[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.

Popraw deklarację i ciało funkcji `initialize_vec_BAD`.

Wskazówka: prawidłowa deklaracja w języku C powinna wyglądać tak:

```
void initialize_vec(int **ptr, int N)
```

* Dla ambitnych

1. Napisz funkcję alokującą pamięć dla tablicy 2-wymiarowej wewnątrz funkcji.
2. Zastanów się, jak wyglądałaby dynamiczna alokacja i zwolnienie pamięci dla tablicy trójwymiarowej – przykładowo gdybyś chciał napisać grę w trójwymiarowe kółko i krzyżyk.