פרק 6
לולאות ומערכים

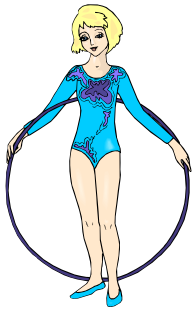
לולאות

- ברצוננו להנחות את התוכנית לבצע קטע קוד מסוים מספר (אולי גדול מאוד) של פעמים.
- כל ביצוע יחיד של קטע הקוד נקרא **איטרציה** (iteration).
- הריצה של הלולאה נשלטת על ידי **תנאי בקרה**. כל עוד תנאי זה מתקיים, הלולאה ממשיכה להתבצע.
- תנאי הבקרה יכול להיות כל ביטוי שניתן לייחס לו ערך אמת (בדומה לתנאי של פקודת **`if`**).

לולאות ב-C

- תנאי הבקרה של הלולאה מחושב ונבדק בין האיטרציות בלבד.
- בזמן שהתוכנית מבצעת את קטע הקוד שבתוך הלולאה, היא אינה בודקת (או מתייחסת בצורה כלשהי) לתנאי. במילים אחרות: גם אם באמצע האיטרציה ערך התנאי משתנה לאפס, הלולאה לא תיפסק אלא כאשר התנאי נבדק שוב (וזאת כמובן בהנחה שבשלב הבדיקה התנאי עדיין אפס).
- בשפת C קיימים 3 סוגי לולאות: **`while`**, **`do-while`** ו-**`for`**.
- כל לולאה ניתן, בעקרון, לממש בעזרת שלושת הסוגים. הבחירה באיזה סוג להשתמש תלויה לרוב בנוחות ובסגנון התכנות.

לולאת `while`



```
while (תנאי)
    פקודה ;
```

```
{
    בלוק
}
```

1. התוכנית בודקת את תנאי הבקרה.
2. אם התנאי מתקיים, מתבצעת איטרציה אחת של הלולאה. אחרת, הלולאה נפסקת והתוכנית ממשיכה אחרי הלולאה.
3. בסוף כל איטרציה התוכנית חוזרת לראש ה-**`while`** ובודקת את התנאי שוב. אם הוא מתקיים מתבצעת איטרציה נוספת, אחרת הלולאה נפסקת.

דוגמה ל-while

- התוכנית הבאה קוראת רצף תווים, ומזהה בתוכו רצפים של ספרות.
התוכנית מחשבת את אורכו של רצף הספרות הארוך ביותר שנקרא:

```
curr_len = 0; max_len = 0;

while((c=getchar()) != EOF)
{
  if (isdigit(c)) {
    curr_len++;
    max_len =
      (curr_len > max_len) ? curr_len : max_len;
  }
  else
    curr_len = 0;
}
```

הפונקציה **isdigit()** מחזירה 1 אם התו שקיבלה מייצג סיפרה, ו-0 אחרת. היא מוגדרת בקובץ הספרייה **<ctype.h>**.

לולאת do-while

```
do
    ; פקודה
while (תנאי);
```

```
{
    בלוק
}
```

1. התוכנית מבצעת איטרציה אחת של הלולאה (איטרציה זו מתבצעת בכל מקרה וללא תלות בתנאי הבקרה).
2. בתום האיטרציה נבדק תנאי הבקרה.
3. אם התנאי מתקיים, התוכנית חוזרת ל-**do** ומבצעת איטרציה נוספת של הלולאה. אחרת, הלולאה נפסקת והתוכנית ממשיכה לאחריה.

דוגמאות ל-do-while

קריאת תווים מ-**stdin** עד סוף הקלט:

```
do {
  c = getchar();
} while (c != EOF);
```

```
i = 3;

do {
  printf("%-2d!\n", i);
} while (--i);

printf("go!\n");
```

הפלט:

```
3 !
2 !
1 !
go!
```


כתיבת קוד עם do-while

- מה עושה הקוד הבא:

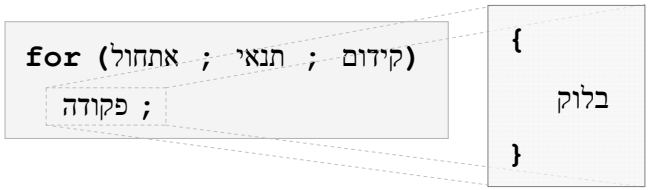
```
while ((c=getchar()) != EOF);
```

- ועכשיו... ?

```
do {
  printf("Thank you for entering %c!\n",c);
}
while ((c=getchar()) != EOF);
```

- מסקנה: פקודת **while** השייכת ל-**do** מעליה יש תמיד לכתוב באותה השורה עם הסוגר המסולסל **}** המסיים את הבלוק!

לולאת for



1. התוכנית מבצעת את האתחול הרשום בפקודת ה-**for**.
2. התוכנית בודקת את תנאי הבקרה. אם הוא מתקיים, מתבצעת איטרציה אחת של הלולאה, אחרת הלולאה נפסקת והתוכנית ממשיכה לאחריה.
3. בסוף כל איטרציה התוכנית חוזרת לראש ה-**for** ומבצעת את הקידום. לאחר מכן התנאי נבדק שוב, ולפי התוצאה התוכנית קובעת אם לבצע איטרציה נוספת, או להפסיק את הלולאה.

דוגמאות ל-for (1)

תהיתם פעם אילו אותיות "זכו" לקבל קוד ASCII ראשוני? הנה:

```
for (c='a' ; c<='z' ; c++)
{
  is_prime = 1;
  for (i=2; i<=(int)sqrt(c); i++) {
    if (c%i == 0)
      is_prime = 0;
  }
  if (is_prime) {
    printf("%c is prime\n", c);
  }
}
```

הפונקציה **sqrt()** מחזירה את השורש של הפרמטר שלה. היא מוגדרת בקובץ הספרייה **<math.h>**.

דוגמאות ל-for (2)

סדרת פיבונאצ'י מתחילה מהמספרים 0 ו-1, וכל איבר נוסף בה שווה לסכום שני האיברים שלפניו. כלומר, הסדרה היא: 0, 1, 1, 2, 3, 5, 8, 13, …

```
long curr = 1, prev = 0, temp;
int n, i;

printf("Enter n: ");
scanf("%d", &n);

for (i=2; i<=n; i++){
  temp = curr;
  curr = curr + prev;
  prev = temp;
}

printf("Your number is: %ld\n", n ? curr:0);
```

Run!

עוד על הפרמטרים של for

```
for (קידום ; תנאי ; אתחול)
  פקודה ;
```

- כל אחד מן הפרמטרים הוא **ביטוי** המקבל ערך.
- אין הכרח לציין את כל שלושת הפרמטרים: אפשר להשאיר את חלקם ריקים. עם זאת, שני תווי ה-**';** המפרידים בין הפרמטרים הכרחיים, וחייבים להופיע תמיד.
- תנאי ריק שקול לערך true, כלומר זו לולאה אינסופית.
- במידת הצורך, כל אחד מן הפרמטרים ב-**for** יכול לכלול מספר ביטויים, על ידי הפרדתם עם אופרטור **,** (פסיק).

דוגמאות ל-for (3)

- לולאה בעלת **n** איטרציות :

```
for ( ; n>0; n--)
{
  printf("Why do I get the feeling ");
  printf("that I've said this before?\n");
}
```

- אתחול וקידום ריקים (שקול ללולאת **while**):

```
for ( ; (c=getchar()) != EOF ; ) {
  putchar(c);
}
```


אופרטור פסיק (,)

ביטוי 2 , ביטוי 1

- אופרטור פסיק מאפשר לקחת זוג ביטויים ולשרשר אותם, כך שזוג הביטויים יחד **נחשב כביטוי אחד**.

- בכל מקום שבו צריך להיות ביטוי יחיד, ניתן לשים את זוג הביטויים עם פסיק ביניהם.
- הביטויים המשורשרים מחושבים משמאל לימין.
- הערך שמחזיר האופרטור פסיק הוא זה של הביטוי האחרון שחושב – כלומר הביטוי הימני.
- אופרטור פסיק הינו האופרטור בעל הקדימות הנמוכה ביותר.

אופרטור פסיק (,)

ביטוי 1 , ביטוי 2 , ביטוי 3 , ... ⟵ **ביטוי 1 , ביטוי 2**

- ביטוי 2 יכול בעצמו להיות זוג ביטויים עם פסיק ביניהם. למעשה, ניתן להמשיך כך ולקבל מספר כרצוננו של ביטויים משורשרים.
- השרשור של כל הביטויים יחד מתפקד כביטוי יחיד.
- האסוציאטיביות של אופרטור פסיק הינה משמאל לימין ולכן הביטויים בשרשרת מחושבים משמאל לימין.
- הערך שמחזיר האופרטור פסיק הוא כאמור זה של הביטוי האחרון שחושב, כלומר ערכו של הביטוי הימני ביותר בשרשרת.

for עם אופרטור פסיק

- אופרטור פסיק מאפשר הכנסת קוד רב לתוך ראש ה-**for**.
- בדוגמה הבאה מופיעה לולאת **for** שמחשבת את מספר הספרות של מספר שלם – כאשר גוף הלולאה ריק לגמרי!
- שימו לב שבדוגמה זו הכנסנו לראש ה-**for** קוד מעבר למינימום הנדרש לבקרת הלולאה. בדרך כלל רצוי להימנע מכך, כיוון שהקוד המתקבל נוטה להיות מבלבל וקשה לקריאה.

```
int num, positive_num, dig_num;
printf("Enter a number: "); scanf("%d", &num);

for ( dig_num = 1, positive_num = abs(num) ;
      positive_num > 9 ;
      positive_num /= 10, dig_num++ )
;
```

דוגמה טובה יותר

- נניח **dig_num** מכיל את מספר הספרות של **positive_num**. נרצה לבדוק האם **positive_num** הוא מספר סימטרי, כלומר האם הוא זהה כאשר כותבים את הספרות שלו בסדר הפוך.
- בדוגמה זו, ראש ה-**for** מכיל אך ורק קוד הקשור בבקרת הלולאה.

Run!

```
is_symmetric = 1;
for (i=0, j=dig_num-1 ; i<j; i++, j--)
{
  dig1 = positive_num/(int)pow(10,i) % 10;
  dig2 = positive_num/(int)pow(10,j) % 10;
  if (dig1 != dig2) is_symmetric = 0;
}
```


מנגנון הבקרה של לולאת for

- מבנה לולאת **for**, בייחוד עם האופרטור פסיק, מאפשר גמישות רבה בעיצוב הלולאה (עד כדי כך שאפשר לכתוב לולאות **for** משמעותיות אפילו עם גוף לולאה ריק לחלוטין!).
- עם זאת, חשוב להבין את הרעיון מאחורי לולאת ה-**for**: המטרה היא לרכז את מנגנון הבקרה של הלולאה בראש הלולאה, ולבודד אותו מהקוד האיטרטיבי עצמו.
  היתרון באופן כתיבה זה הוא שדי בהתבוננות קצרה בראש הלולאה כדי להבין את התנהגותה.

- תכנות החורג מעקרון זה בדרך כלל קשה להבנה, ואיננו מומלץ.

הפקודה break

- הפקודה **break** מורה על הפסקה מיידית של ביצוע לולאה או פקודת **switch**.
- הפקודה מתייחסת תמיד ללולאה או ה-**switch** הפנימיים ביותר.
- הדוגמה הבאה מחקה את פעולת הפונקציה **scanf("%d")**, שקוראת מהקלט רצף ספרות ומתרגמת אותו למספר שלם:

```
num = 0;
while ( (c=getchar()) != EOF ) {
  if (!isdigit(c)) {
    break;
  }
  num = num*10 + (c - '0');
}
```


קירוב ערכו של π

- ניתן לחשב קירוב לערכו של π בעזרת הנוסחה: $\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$
- הנוסחה היא **אינסופית**, וככל שמחברים בה יותר איברים, מקבלים תוצאה קרובה יותר ל-π.
- בכל איטרציה נחבר איבר נוסף בסדרה. נעצור כאשר השינוי בתוצאה נעשה קטן מאוד, כך שהוא יורד בערכו המוחלט מתחת לסף מסוים.

Run!

```
pi = 0; i = 1; sign = 1;
while (1) {
  delta = 4 * (1.0/i);
  pi += sign*delta;
  if (delta < 1e-6)
    break;
  i += 2; sign *= -1;
}
```

הפקודה continue

- הפקודה **continue** גורמת להפסקת האיטרציה הנוכחית של הלולאה, בדיקת התנאי, והמשך הלולאה באיטרציה הבאה (בהנחה שהתנאי מתקיים).
- במקרה של לולאת **for**, פעולת הקידום שצריכה להתבצע לפני האיטרציה הבאה מתבצעת כרגיל (פעולת הקידום מייצגת בעצם את מה ש'מתקדם' בין האיטרציות; לכן, מהגדרת הלולאה **for**, כדי לעבור לאיטרציה הבאה יש לבצע ראשית את הקידום שהוגדר בראש הלולאה).

תוכנית המחשבת את ממוצע המספרים החיוביים בקלט:

```
sum = 0; count = 0;
while (scanf("%d", &num)==1) {
  if (num < 0)
    continue;
  sum += num; count++;
}
average = sum / count;
```


מי הג'יני הרע?

```
for (i = 0; i<3; ++i) {
  printf("make a wish: ");
  scanf("%s", the_wish);
  if (toohard(the_wish))
    continue;
  dowish(the_wish);
}
```

```
i = 0; while (i<3) {
  printf("make a wish: ");
  scanf("%s", the_wish);
  if (toohard(the_wish))
    continue;
  dowish(the_wish);
  i++;
}
```


שיפור תוכנית האותיות הראשוניות

```
for (c='a' ; c<='z' ; c++)
{
   is_prime = 1;
   if (c%2 == 0) continue;
   for (i=3; i <= (int)sqrt(c); i+=2) {
      if (c%i == 0) {
         is_prime = 0;
         break;
      }
   }
   if (is_prime)
      printf("%c is prime\n", c);
}
```

- שימו לב ל-**break**, שמתייחס ללולאת ה-**for** הפנימית בלבד.

חישוב or לוגי של n ערכים

- את הקוד הבא אפשר לתכנת בצורה פשוטה יותר, אבל נעזר בו לשם ההדגמה.
- נבצע חישוב or באמצעות לוגיקה מקוצרת: כל עוד נקראים אפסים, נגיע ל-**continue** ונמשיך לקרוא ערכים. ברגע שנראה ערך שונה מאפס, נעצור ונחזיר 1.
- איזו תשובה יש להחזיר בסוף? אם עצרנו מוקדם (כלומר **i<n**), סימן שהתשובה היא "1". אם לא, סימן שהתשובה "0".
- שימו לב שבתוך ה-**switch**, פקודת ה-**break** מתייחסת ל-**switch** בלבד, ואילו פקודת ה-**continue** מתייחסת ל-**for**.

```
for (i=0; i<n; ++i) {
  scanf("%d", &num);
  switch(num == 0) {
    case 0: break;
    case 1: continue;
  }
  break;
}
result = (i < n);
```

מערכים

- **מערך** הוא אוסף סדור של משתנים מאותו הטיפוס.
- בשפת C ניתן ליצור מערך מכל טיפוס, בסיסי או מורכב.
- מספר המשתנים שהמערך מכיל נקרא **אורך המערך**.
- מצהירים על המערך יחד עם יתר המשתנים בתוכנית. בזמן ההצהרה יש לקבוע גם אורך המערך.

```
int array[10];
```

- שורה זו מצהירה על מערך ששמו **array**. מערך זה מכיל **10** איברים, כולם מטיפוס **int**.
- מערך איננו נחשב טיפוס בסיסי ב-C, אלא טיפוס מורכב.

דוגמאות למערכים

- מערך מטיפוס **double**, באורך 5:

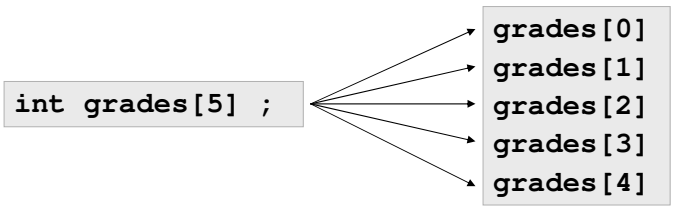
```
double grades[5] ;
```

- מקובל לקבוע את אורך המערכים כקבועים בראש הקובץ, באמצעות **#define** :

```
#define N 50
#define SEQUENCE_LEN 1000

int main()
{
  double salaries[N];
  double sequence[SEQUENCE_LEN];
  …
}
```


איברי המערך

- איברי המערך ממוספרים, ויש ביניהם סדר. עבור מערך בגודל **N**, האיבר הראשון מספרו **0** והאחרון מספרו **N-1**.
- ניתן לגשת ישירות לכל איבר במערך, ולעבוד איתו כמשתנה עצמאי לכל דבר.
- על מנת לגשת לאיבר בעל אינדקס **k** במערך **grades** יש לרשום **grades[k]**. שימו לב שזהו האיבר ה-**k+1** במערך.



דוגמאות לשימוש במערכים

- נמלא את המערך:

```
grades[0] = 95;
grades[1] = 45;
grades[2] = 62;
grades[3] = 80;
grades[4] = 76;
```

- נדפיס את תוכנו :

```
printf("grade 0: %d", grades[0]);
printf("grade 1: %d", grades[1]);
printf("grade 2: %d", grades[2]);
 …
```

- נסכום אותו:

```
sum = grades[0] + grades[1] +
      grades[2] + grades[3] +
      grades[4] ;
```

האופרטור [ ]

- הסוגריים המרובעים **[ ]** הם אופרטור בינארי: הם מקבלים שני פרמטרים, הראשון הוא מערך והשני מספר טבעי.
- עבור מערך **A** ואינדקס **k**, הביטוי **A[k]** הוא האיבר עם אינדקס **k** במערך. כפי שראינו, איבר זה מתפקד כמשתנה לכל דבר.
- חשוב: בשפת C לא מתבצעת בדיקת טווח עבור האינדקס **k**. כלומר, אם נציין אינדקס החורג מהטווח (מספר שלילי, או גדול מ-**N−1**), לא נקבל כל שגיאת קומפילציה, ובסיכוי טוב אפילו לא שגיאת ריצה. למשל, השורות הבאות חוקיות ב-C (על אף שהן ייגרמו להתנהגות בלתי צפויה בזמן ריצת התוכנית):

```
grades[-1] = 100;
grades[1000] = 5;
```


לולאות ומערכים

- בתוך הסוגריים המרובעים **[ ]** אין הכרח לשים מספר קבוע, אלא ניתן לרשום **כל ביטוי** שטיפוסו מספר שלם. בזמן ריצת התוכנית ביטוי זה מחושב, ועל פיו מתבצעת גישה לתא המתאים בזיכרון.
- בזכות תכונה זו, לולאות נעשות לכלי עבודה טבעי עם מערכים.
- למשל, נמלא מערך באמצעות לולאה:

```
for (i=0; i<N; ++i)
{
  printf("What salary were you thinking of? ");
  scanf("%lf", &salaries[i]);
  salaries[i] /= 2;
}
```


לולאות ומערכים

- חישוב המשכורת הממוצעת והמשכורת המקסימאלית:

```
max_sal = salaries[0];

for (sum=0, i=0; i<N; ++i)
{
  sum += salaries[i];
  if (salaries[i] > max_sal)
    max_sal = salaries[i];
}

average_sal = sum / N;
```

סוגי פקטורים בטכניון

- פקטור כפל:

```
for (i=0; i<5; ++i)
  grades[i] = (int)(grades[i] * 0.8);
```

- פקטור שורש:

```
for (i=0; i<5; ++i)
  grades[i] = (int)(sqrt(grades[i]) * 10);
```

- פקטור שיבֵּר:

```
for (i=0; i<5; ++i)
  if (grades[i] < 54) grades[i] = 54;
```

אתחול מערכים

- כמו משתנים אחרים, גם מערכים ניתן לאתחל בזמן שמצהירים עליהם.
- מערך שאינו מאותחל עשוי להכיל תוכן כלשהו (= זבל) בזמן שהוא נוצר.
- אתחול מערך נעשה על ידי ציון רשימת האיברים שלו (**רשימת אתחול**):

```
int grades[5] = { 95, 45, 62, 80, 76 };
```

- אם הרשימה אינה מכילה מספיק ערכים, אז הערכים שברשימה ממלאים את תחילת המערך, ויתר המערך ממולא באפסים:

```
int grades[5] = { 95, 45 };
```

- הפקודה הבאה מאתחלת את כל המערך לאפסים:

```
int grades[5] = { 0 };
```


אתחול מערכים

- כאשר משתמשים ברשימת אתחול, ניתן גם להשמיט את אורך המערך.
- במקרה זה אורך המערך יבחר אוטומטית להיות כאורך רשימת האתחול.

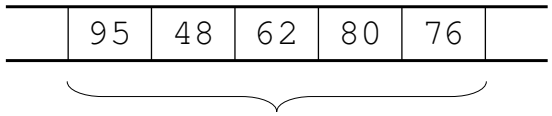
```
char cards[] = { '2','3','4','5','6',
    '7','8','9','X','J','Q','K','A' };
```

שקול

```
char cards[13] = { '2','3','4','5','6',
    '7','8','9','X','J','Q','K','A' };
```


אחסון המערך בזיכרון

- תוכן המערך מאוחסן בזיכרון באופן **סדרתי ורציף**. כלומר, אברי המערך נמצאים בזיכרון בזה אחר זה, לפי סדר האינדקסים שלהם:



- לדוגמה, אם גודלו של **int** הוא ארבעה בתים, אזי כמות הזיכרון שיצרוך המערך **grades** כולו היא 20 בתים. ארבעת הבתים הראשונים יכילו את השלם הראשון, הארבעה שלאחריהם את השלם השני, וכן הלאה.

מערכים דו-ממדיים

- מערך דו-ממדי דומה למערך חד-ממדי, אך ניגשים בו לכל איבר על ידי ציון שני אינדקסים. הצהרה על מערך דו ממדי ב-C נראית כך:

```
int array2d[N][M];
```

- הצהרה זו מקצה זיכרון למערך דו-ממדי בשם **array2d**. על מנת לגשת לאיבר כלשהו במערך זה, יש לציין שני אינדקסים כך:

```
array2d[i][j]
```

- לכל $\begin{cases} 0 \le i \le N-1 \\ 0 \le j \le M-1 \end{cases}$ ישנו איבר מתאים במערך. לכן, בסך הכל המערך מכיל N*M איברים, כאשר במקרה זה כל אחד מהם מטיפוס **int**.

דוגמה למערך דו-ממדי

- נניח **STUD_NUM** תלמידים, שלכל אחד מהם **GRAD_NUM** ציונים. הציונים מאוחסנים במערך **grades**, כאשר **grades[i][j]** הוא הציון ה-**j** של התלמיד ה-**i**. הקוד הבא מחשב את הציון הממוצע של כל תלמיד, וכותב אותו למערך **averages**:

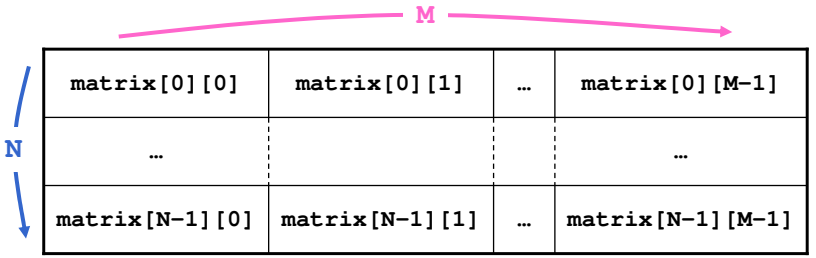
```
double grades[STUD_NUM][GRAD_NUM];
double averages[STUD_NUM];
int i, j;

for (i = 0; i < STUD_NUM; ++i) {
  averages[i] = 0;
  for (j = 0; j < GRAD_NUM; ++j) {
     averages[i] += grades[i][j];
  }
  averages[i] /= GRAD_NUM;
}
```


מערך דו-ממדי כמטריצה

- נתבונן במערך הדו-מימדי הבא (**N** ו-**M** מוגדרים ב-**#define**):

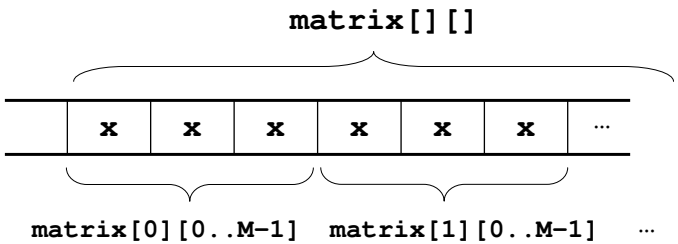
```
double matrix[N][M];
```

- נוח לחשוב על מערך זה כמטריצה דו-ממדית עם **N** שורות ו-**M** עמודות, ולשרטט את אברי המערך הדו-מימדי כך:



אחסון מערך דו-ממדי בזיכרון

- הזיכרון במחשב הוא ליניארי, ולכן יש לאחסן את אברי המערך הדו-ממדי בזיכרון בזה אחר זה, בסדר כלשהו.
- הסדר בו מאוחסנים אברי מערך דו-ממדי בזיכרון הינו שורה-אחר-שורה. כלומר, שורות המטריצה פרושות בזיכרון בזו אחר זו, כך:



אתחול מערך דו-ממדי

- אתחול מערך דו-ממדי נעשה על ידי ציון רשימת איבריו, לפי הסדר בו הם מופיעים בזיכרון (כלומר שורה-שורה):

```
int matrix[2][3] = { 5, 3, 8, 1, 0, 3 };
```

- לשם הנוחות, ניתן גם לחלק את רשימת האיברים לפי שורות, כך:

```
int matrix[2][3] = { {5, 3, 8} ,
                     {1, 0, 3} };
```

- מה לגבי האפשרות לתת רשימת אתחול בלי לציין את גודל המטריצה, כפי שעשינו במקרה החד-ממדי?

אתחול מערך דו-ממדי

- נניח נכתוב שורה כזו:

```
int matrix[][] = { 5, 3, 8, 1, 0, 3 };
```

- בכתיבה זו ממדי המטריצה אינם מוגדרים היטב, והתוכנית לא תוכל לדעת לאיזו אפשרות התכוונו! ישנן הרבה אפשרויות במקרה זה, למשל: **`matrix[6][1]`**, **`matrix[1][6]`**, **`matrix[2][3]`** ...

- על מנת לפתור את אי הבהירות, הדרישה היא תמיד לציין לפחות את אורך השורה של המטריצה, כך:

```
int matrix[][3] = { 5, 3, 8, 1, 0, 3 };
```

מכאן הקומפיילר יכול לחשב לבד את מספר השורות (2).