# תכנות מערכות א
# תרגול 6



© TemplatesWise.com

# תכניה

- מצביעים
- מצביעים ומערכים

# מצביעים



© TemplatesWise.com

# תזכורת- כתובות זיכרון


- הזיכרון במחשב מחולק לתאים, כל אחד בגודל בית אחד.
- תאי הזיכרון ממוספרים. מס' התא נקרא גם **הכתובת** של התא.

# כתובות זיכרון

- לכל תוכנית מוקצה איזור בזיכרון בו היא יכולה לאכסן את נתוניה. אוסף כתובות זה נקרא **מרחב הכתובות** של התוכנית
- כל משתנה מאוחסן באחד או יותר תאי זיכרון רצופים. **כתובת המשתנה** היא הכתובת של התא הראשון שמכיל אותו.
- כדי לקרוא מהזיכרון על התוכנית לציין את הכתובת הרצויה ואת מס' הבתים שיש לקרוא (מס' הבתים שהמשתנה תופס)

# מצביעים

- מצביעים הם משתנים המחזיקים כתובות זיכרון של משתנים אחרים.
- הצהרה על מצביע:

מיקום הכוכבית לא משנה להגדרה

```
int* iptr;
double *dptr;

char* cptr, c;
char *cptr, c;
```

הצהרה על מצביע לint

הצהרה על מצביע לdouble

מה הטיפוס של המשתנה cptr? מה הטיפוס של c?

על מנת למנוע בלבול נכתוב את השורה כך, עם * ליד המשתנה שהוא מצביע

# אופרטור &

- אופרטור & מחזיר את כתובתו של משתנה:

```
&var_name
```

- דוגמא:

```
int* iptr;
double* dptr;

int a = 3;
double d = 6.4;

iptr = &a;
dptr = &d;
```

| | | |
|---|---|---|
| 2000 | d | 6.4 |
| 2008 | a | 3 |
| 2012 | dptr | 2000 |
| 2020 | iptr | 2008 |

# מצביעים

```
int* iptr;
double* dptr;

int a = 3;
double d = 6.4;

iptr = &a;
dptr = &d;
```

- שימו לב! מצביעים הם משתנים לכל דבר, ולכן הם מאוחסנים במחסנית כמו כל משתנה אחר.
- כמות הבתים הדרושה לאחסון כתובת זיכרון **היא זהה לכל סוגי המצביעים** ללא תלות בטיפוס עליו מצביעים

| | | |
|---|---|---|
| 2000 | d | 6.4 |
| 2008 | a | 3 |
| 2012 | dptr | 2000 |
| 2020 | iptr | 2008 |

כמות הבתים הדרושה לאחסון כתובת זיכרון תלויה בארכיטקטורה של המחשב עליו רצה התוכנית.
ברוב המחשבים כיום מצביע תופס 8 בתים (מערכות 64 ביט).

# אופרטור *

- בהינתן מצביע אופרטור * מחזיר את המשתנה שעליו הוא מצביע:

```
*ptr_name
```

- דוגמא: מצביע למשתנה מטיפוס int

```
int a = 3, b = 5;
int* ptr;

ptr = &a;
*ptr = 8;
b = *ptr + 3;
```

הטיפוס של ptr הוא *int

*ptr מתנהג כמו משתנה מטיפוס int

| | | |
|---|---|---|
| 2000 | ptr | 2008 |
| 2004 | b | 11 |
| 2008 | a | 8 |

# מתי נשתמש במצביעים?

- מתי נעביר מצביעים כפרמטר לפונקציה?
  - נעביר כתובת של משתנה לפונקציה כאשר נרצה **לשנות את ערכו של המשתנה-** כך אנו מקנים לפונקציה גישה ישירה לזיכרון המשתנה
    דוגמא:

```
void swap(int* x, int* y)
```

  - ב-C לכל פונקציה ערך החזרה אחד. אם נרצה **פונקציה שמחזירה יותר מערך החזרה אחד**, נממש זאת באמצעות מצביעים.
    דוגמא: פונקציה עם החתימה:

```
int foo(int x, int* res)
```

    יכולה להחזיר 2 ערכים.
    ערכו של אחד מהם יוחזר כערכו של המשתנה res.

# מתי נשתמש במצביעים? (המשך)

– בשפת C ניתן להגדיר טיפוסים התופסים זיכרון רב (למשל מערכים). במקרה זה העברתם לפונקציה by value (כלומר ע"י העתקה) תהיה פעולה כבדה מאוד. ניתן לחסוך את עבודת ההעתקה על ידי **העברת הכתובת של משתנים מטיפוסים כאלה** לפונקציה, במקום את התוכן.

# Swap

- פונקציה המחליפה בין התוכן של שני מספרים מטיפוס int:

```
void swap(int* p, int* q)
{
   int tmp = *p;
   *p = *q;
   *q = tmp;
}
```

מה היה קורה אם לא היינו משתמשים במצביעים?

- נשתמש בפונקציה כך:

```
int a = 3, b = 7;
swap(&a, &b);
printf("a=%d, b=%d", a, b);
```

```
a=7, b=3
```

# מצביעים ופונקציות

- תרגיל 1: כתבו פונקציה המקבלת כקלט אורך רדיוס של מעגל (לא בהכרח שלם) ומחזירה את שטח המעגל והיקפו

```
#include <stdio.h>
#define PI 3.14

void calc_perimeter_and_area(double radius, double* p_perim,
  double* p_area)
{
    *p_perim = 2 * PI * radius;
    *p_area = PI * (radius * radius);
}
```

# מצביעים ופונקציות

- <u>תרגיל 2:</u> כתבו פונקציה המקבלת 3 מצביעים a, b, ו-c, למספרים שלמים, ומחליפה את תוכנם כך שיהיו מסודרים בסדר עולה

# מצביעים ופונקציות

```
#include <stdio.h>

void swap_int(int* p, int* q)
{
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

void sort3(int* a, int* b, int* c)
{
    if (*a > *b) swap_int(a, b);
    if (*a > *c) swap_int(a, c);
    if (*b > *c) swap_int(b, c);
}
```

# כתובת 0 והקבוע NULL

- כאשר משתני רגילים אינם מאותחלים הם מכילים זבל.
- מה קורה כאשר ניגשים לפוינטרים לא מאותחלים?

```
char *ptr;
*ptr='A';
```

- מצביע לא מאותחל מצביע למקום **שרירותי** בזיכרון

# כתובת 0 והקבוע NULL

- הכתובת **0** הינה כתובת מיוחדת, שאינה כתובת חוקית בזיכרון, ומשמשת לאתחול מצביע כאשר רוצים לציין שהוא אינו מאותחל לזיכרון חוקי כלשהו.
- במקום הערך 0, יש להשתמש בקבוע **NULL** – זהו **#define** שמוגדר אוטומטית כאפס.
- מנגנון זה מאפשר לבדוק את חוקיותו של כל מצביע לפני השימוש בו:

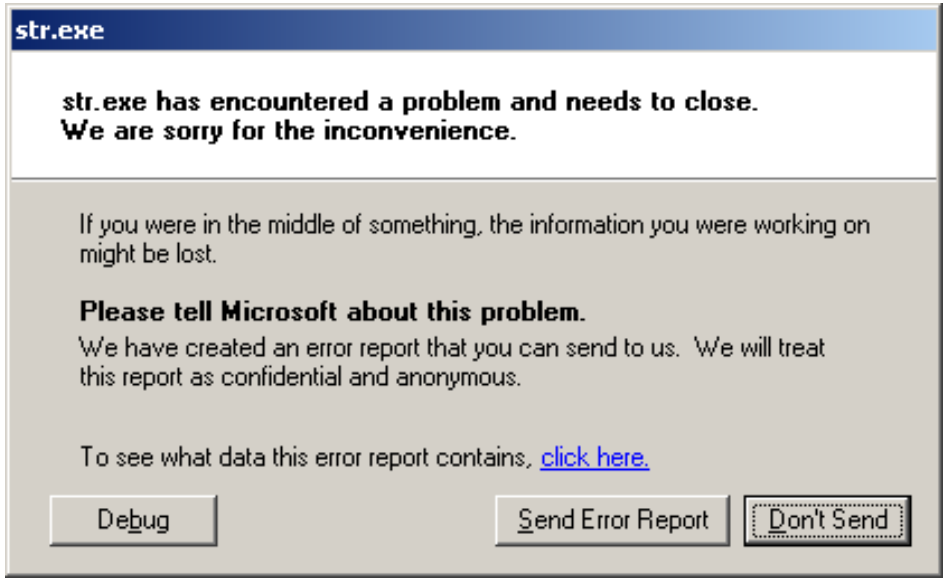
```
if (p != NULL) {
  *p = …
}
else {
  printf("p is unallocated!\n");
}
```

יש לאתחל ל-NULL כל מצביע שאינו מאותחל לזיכרון חוקי, וכן לשים NULL במצביע במידה והזיכרון שאליו הוא מצביע מפסיק להיות מוקצה

# כתובת 0 והקבוע NULL

- ניסיון לגשת לתוכן של מצביע NULL גורר מיידית שגיאת זמן ריצה:

```
int *p = NULL;
*p = 3;
```



# מצביעים- סיכום ביניים

- מצביעים מאפשרים גישה ישירה לזיכרון של משתנה
- שימושים:
  - שינוי הערך של משתנה בתוך פונקציה
  - ריבוי ערכי החזרה מפונקציה
  - העברת משתנים גדולים לפונקציה בלי לבצע העתקה
- טיפוסים שונים למצביעים על פי סוג המידע שהם מצביעים עליו
  - int*, char*, float*, etc.
  - void*
- אתחול מצביעים ל-NULL (כתובת 0) אם אינם מצביעים לכתובת חוקית

# מערכים ומצביעים
# אריתמטיקה של מצביעים



© TemplatesWise.com

# מערכים כפרמטר לפונקציה

- מה ידפיס קטע הקוד הבא?

```
#define N 3
int salaries[N] = {3000, 8500, 5500};

printf("Old salaries:\n");
for (i=0; i<N; ++i) {
  printf("%d\n", salaries[i]);
}

cutSalaries(salaries, N);

printf("New salaries:\n");
for (i=0; i<N; ++i) {
  printf("%d\n", salaries[i]);
}
```

```
void cutSalaries (int salary_array[], int len)
{
  for (i=0; i<len; ++i) {
      salary_array[i] /= 2;
  }
}
```

# מערכים כפרמטר לפונקציה

- הפונקציה שינתה את תוכן המערך!
- ב-C מעבירים משתנים by value, אז איך ייתכן שהמערך שונה והערך החדש מופיע מחוץ לפונקציה בה התבצע השינוי?
- כמו כן, למה היינו צריכים להעביר את אורך המערך לפונקציה?

בשקפים הבאים נסביר מה קורה מאחורי הקלעים כאשר מעבירים מערך לפונקציה. לשם כך נציג אריתמטיקה של מצביעים.

# מערכים בזיכרון

- לפני שנסביר איך משנים מערכים בתוך פונקציה, נרצה לדעת בתור התחלה איך שמורים מערכים בזיכרון:
- מערך יישמר ברצף של תאים בזיכרון. לדוגמא מערך של int:

```
int arr[] = {1,-3,5,7,8,4}
```

מס' התא

| #1000 | #1004 | #1008 | #1012 | #1016 | #1020 |
|---|---|---|---|---|---|
| 1 | -3 | 5 | 7 | 8 | 4 |

- מטריצה או מערך דו ממדי שמורים בזיכרון גם כן ברצף, שורה אחרי שורה. לדוגמא במערך דו ממדי של char:

```
char mat[][3] =
    {{'a', 'b', 'c'},
     {'d', 'e', 'f'},
     {'g', 'h', 'i'}};
```

| #100 | #101 | #102 | #103 | #104 | #105 | #106 | #107 | #108 |
|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | f | g | h | i |

# אריתמטיקה של מצביעים

- אריתמטיקת מצביעים מאפשרת לקדם מצביעים, וכן לחבר אליהם מספרים שלמים. ניתן לעשות שימוש בכל האופרטורים החיבוריים:

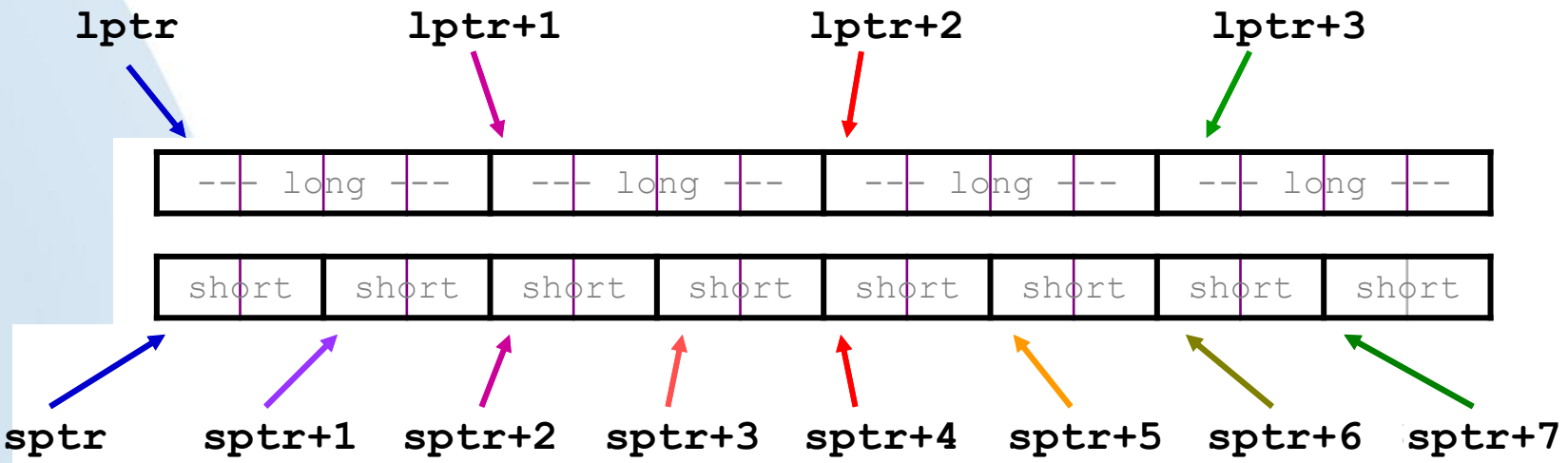
  `+` `-` `+=` `-=` `++` `--`

- האריתמטיקה מוגדרת כך: כל תוספת/הפחתה של 1 משנה את הכתובת שרשומה במצביע במספר בתים **כגודל הטיפוס אליו מצביעים**.
- ניתן לחסר שני מצביעים זה מזה, רק אם הם **מאותו טיפוס**. התוצאה מומרת למספר שלם (`int`) המחזיק את המרחק ביניהם בזיכרון.
- אסור לחבר/להכפיל/לחלק שני מצביעים או להכפיל/לחלק מצביע בקבוע.

# אריתמטיקה של מצביעים

- משתמשים באריתמטיקת מצביעים כאשר יש לנו בזיכרון מספר איברים רצופים מאותו הטיפוס (למשל במערך).
- בהינתן מצביע לאחד מהאברים בזיכרון, הוספת 1 אליו תקדם אותנו **לאיבר הבא בזיכרון**, ואילו הפחתת 1 תחזיר אותנו לאיבר הקודם.

```
long *lptr;    short *sptr;
```



# השוואה בין מצביעים

- בשפת C, ניתן להשוות בין שני מצביעים על ידי השוואת הכתובות שהם מכילים. מצביע **ptr1** הוא **קטן** ממצביע **ptr2** אם הוא מכיל כתובת מוקדמת יותר בזיכרון.
- השוואה בין מצביעים מועילה, למשל, כאשר יש לנו שני מצביעים לאותו המערך, ואנו רוצים לדעת איזה מהם מצביע למקום קודם במערך. במקרה זה, אם מצביע אחד קטן מהשני – סימן שהוא מצביע לתא מוקדם יותר במערך.
- ניתן להשתמש בכל אחד מן האופרטורים הבאים להשוואת מצביעים:

**!=  ==  <=  >=  <  >**

# מערכים כמצביעים

- בשפת C, כתיבת שמו של מערך כלשהו ללא האופרטור `[]` מחזירה אוטומטית **מצביע לאיבר הראשון במערך**.
- אם נניח שטיפוס המערך הוא `T[N]`, אזי טיפוס המצביע המוחזר יהיה `T*`.

```
int speeds[] = { 25, 50, 80, 90, 110 };

printf("%p", &speeds[0]);
printf("%p", speeds);
```

דגל להדפסת מצביע

הביטויים שקולים

# מערכים כמצביעים

- איך נפנה לאיבר באינדקס 3 במערך speeds?

```
int speeds[] = { 25, 50, 80, 90, 110 };
```

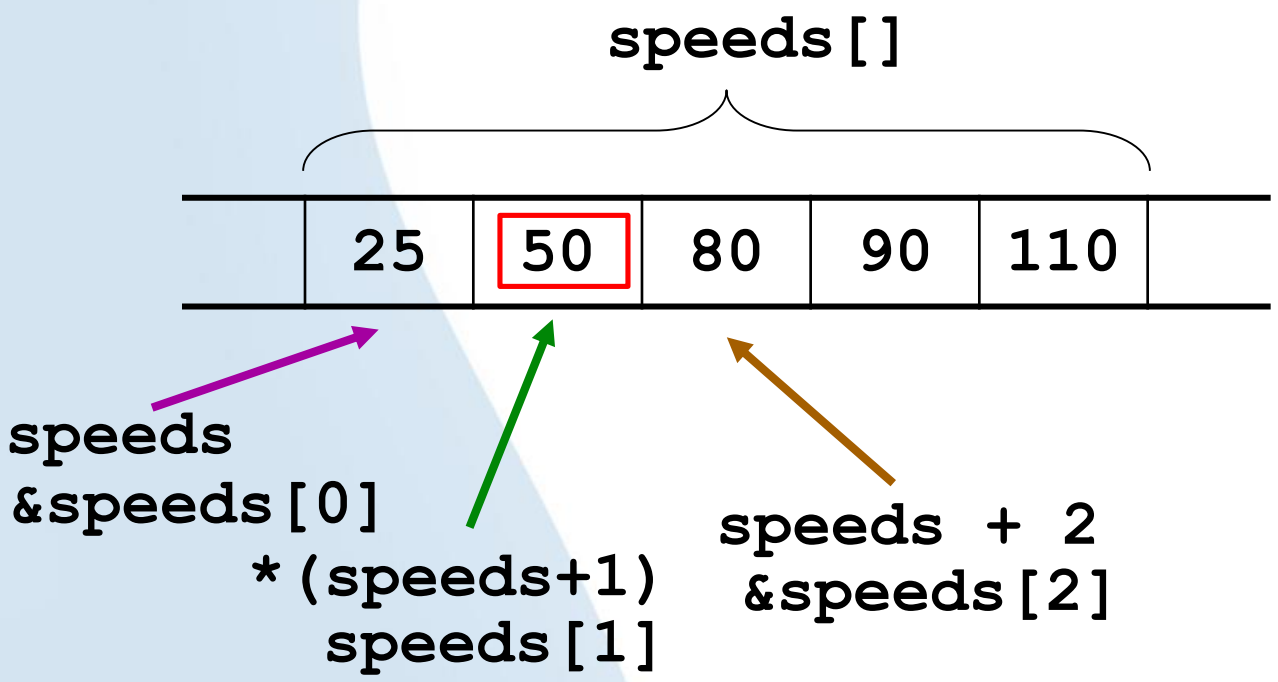
- תזכורת: איברי המערך נמצאים בזיכרון בצורה **רציפה**.
- כיוון שיש לנו מצביע לתחילת המערך, ניתן באמצעות **אריתמטיקה של מצביעים** להגיע לאיבר **הרביעי** כך:

```
printf("%p", speeds+3);
*(speeds+3) += 10;
```

נדפיס את כתובת האיבר הרביעי

נשנה את תוכנו של האיבר הרביעי

# מערכים כמצביעים




עבור מערך בשם a

| כתיב מצביעים | כתיב מערכים |
|---|---|
| a | &a[0] |
| a+i | &a[i] |
| *(a+i) | a[i] |

# מערכים כמצביעים

- תרגיל 3: כתבו פונקציה המקבלת מערך שלמים ומצביע לאיבר האחרון שבו ומחזירה את אורכו.

```
int get_array_length(int a[], int *p_last)
{
    return p_last - a + 1;
}
```

# מצביעים לעומת מערכים- השוואה

- אילו הבדלים מהותיים בין מצביע למערך?

1. שמו של מערך הוא קבוע, כלומר לא ניתן לשנות את הכתובת שאליה הוא מצביע.

```
int a[5], b[8];
a = b;
```

שגיאת קומפילציה

לעומת זאת מצביע יכול להכיל מס' כתובות שונות לאורך ריצת התוכנית:

```
int a, b, *ptr;
ptr = &a;
ptr = &b;
```

# מצביעים לעומת מערכים- השוואה

- אילו הבדלים מהותיים בין מצביע למערך?

2. יצירת מערך מקצה זיכרון לאברי המערך. לעומת זאת מצביע זקוק לזיכרון שמוקצה על ידי גורם חיצוני.

```
int *ptr;
ptr[0] = 100;
```

שגיאת warning. גם כן יתכן שגיאת זמן ריצה בהנחת שכתובת הזבל שמכיל ptr אינה חוקית

# מצביעים לעומת מערכים- השוואה

- אילו הבדלים מהותיים בין מצביע למערך?

3. עבור משתנה שהוא מערך – כמו כל משתנה אחר – התוכנית יודעת כמה זיכרון הוא תופס, כיוון שזה מוגדר בטיפוס שלו. לכן, ניתן לקבל את גודלו של המערך בזיכרון באמצעות האופרטור **sizeof**.
לעומת זאת, מצביע איננו יודע על איזה אורך מערך הוא מצביע, והפעלת **sizeof** עליו פשוט מחזירה את כמות הזיכרון הדרושה לאחסון המצביע עצמו.
למשל, ב-Code::blocks נקבל:

```
int a[5], *p;
p = a;
printf("%d %d", sizeof(a), sizeof(p));
```

20 8

# מערכים כפרמטר לפונקציה

- בתחילת הפרק ראינו שהפונקציה יכולה לשנות ערכים בתוך מערך, שערכים אלה "שורדים" גם מחוץ לפונקציה
- כידוע בשפת C פרמטרים עוברים לפונקציה by value
- עכשיו יש בידינו את הכלים ליישב את הסתירה:

# החתימה של פונקציה המקבלת מערך

- למעשה החתימה הבאה:

```
double average(int grades[], int n);
```

- **שקולה לחלוטין**, מכל בחינה שהיא, לחתימה הבאה:

```
double average(int* grades, int n);
```

- בשני המקרים יש לקרוא לפונקציה כמו בקריאה הבאה:

```
int grades[3]={90,98,65};
double res = average(grades,3);
```

- והטיפוס של הפרמטר הראשון בשני המקרים יהיה `int*`!
- צורת הכתיבה הראשונה היא פשוט צורת כתיבה ש-C מאפשרת להשתמש בה כשרוצים להדגיש כי הכוונה שהפונקציה תקבל מערך כפרמטר, ולא סתם מצביע ל-**`int`** בודד. ניתן להשתמש בצורת כתיבה זו <u>רק עבור פרמטרים של פונקציות</u>! לא ניתן להשתמש בה בהצהרות על משתנים, למשל.

# מערכים כפרמטר לפונקציה

- המשמעות היא שלמעשה בשפת C **מערך אינו יכול להיות פרמטר לפונקציה**
- הדרך היחידה להעביר מערך לפונקציה היא על ידי <u>העברת הכתובת שלו</u>. עושים זאת באמצעות שימוש בשם המערך המתפקד גם כמצביע לאיבר הראשון בו.
- מעשית, צורת העברה זו טובה יותר!
  זאת כיוון שמערך עשוי לתפוש זיכרון רב, ועדיף להימנע משכפול תוכנו בכל פעם שמעבירים אותו כפרמטר לפונקציה.

# מערך כפרמטר לפונקציה

- לסיכום, פונקציה המקבלת מערך תראה כך:

```
void print(int *array, int len)
{
   for (int i=0; i<len; ++i)
     printf("%d\n", array[i]);
}

int my_array[8];
print(my_array, 8);
```

הפונקציה תקבל מצביע בתור פרמטר

כיוון שקיבלנו מצביע (ולא מערך) איננו יודעים מה אורכו של המערך ולכן נעביר אותו כפרמטר נפרד לפונקציה

בתוך הפונקציה נעבוד עם המצביע כמו שהיינו עובדים עם מערך

בקריאה לפונקציה נציין את שם המערך (שיהפוך אוטומטית למצביע לתחילת המערך)

# מערכים כפרמטר לפונקציה

- <u>תרגיל 4:</u> כתבו פונקציה המקבלת מערך של מצביעים ל- int ומחזירה את מספר המצביעים שאינם בשימוש.
- כתבו פונקציית main המשתמשת בפונקציה שלכם בצורה תקינה

# מערכים כפרמטר לפונקציה

```
#include <stdio.h>

int get_non_used_ptrs(int* a[], int len)
{
    int i, count = 0;

    for (i = 0; i < len; i++)
        if (a[i] == NULL)
            count++;

    return count;
}
```

# מערכים כפרמטר לפונקציה

```
int main()
{
    int x = 3, y = 6;
    int* arr[] = {&x, NULL, &y, NULL, NULL};

    printf("Non-used pointers in array: %d\n",
                                get_non_used_ptrs(arr, 5));

    return 0;
}
```

# מערכים כפרמטר לפונקציה

- נשים לב שלמעשה הפונקציה לא קיבלה עותק זמני של המערך' אלא אנו מספקים לה גישה ישירה למערך שלנו. כל שינוי שהיא תבצע במערך יתרחש במערך המקורי עצמו.
- למשל, הפונקציה הבאה מעדכנת ציונים באמצעות פקטור חיבורי:

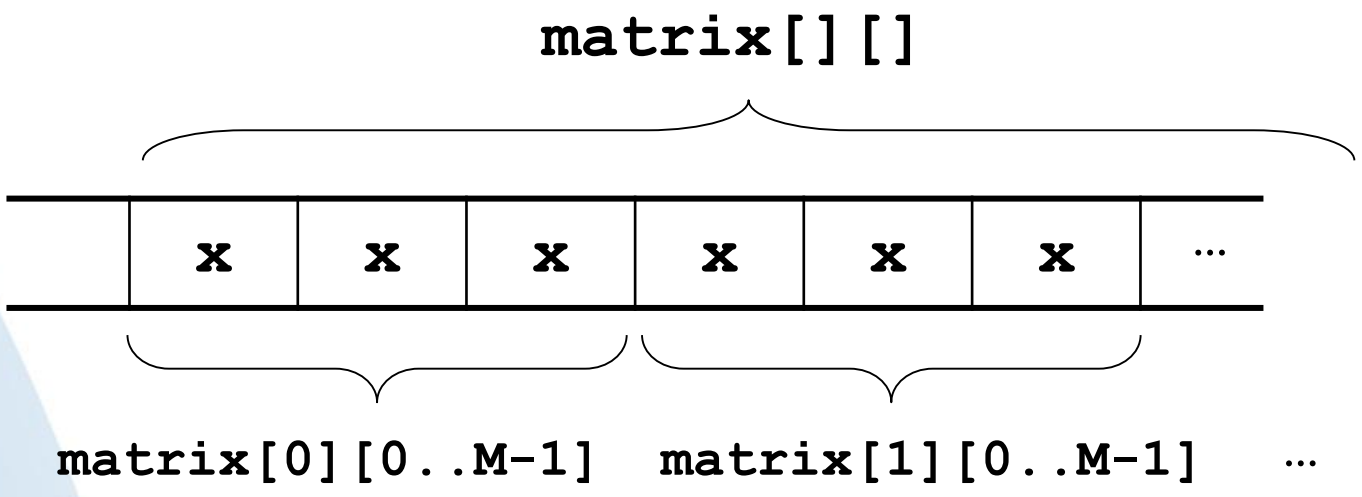
```
void factor(int *grades, int n)
{
  for (int i=0; i<n; ++i) {
    grades[i] += 5;
  }
}
```

# העברת מערך דו מימדי לפונקציה

- נגדיר מערך דו ממדי:

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

- מערך זה פרוש בזיכרון ברצף שורה אחר שורה:



# העברת מערך דו ממדי לפונקציה

- כיצד נחשב מצביע לאיבר `matrix[i][j]` ? נבצע זאת בשלבים:
- כתובת תחילת המערך הדו-ממדי היא:

```
&matrix[0][0]
```

- כעת נמצא את כתובתו של האיבר הראשון בשורה ה- `i` הראשון, כלומר מצביע ל-`matrix[i][0]` :
נשים לב ששורה זו היא למעשה השורה ה-`i+1` במטריצה, ולכן יש `i` שורות לפניה. כיוון שבכל שורה יש `M` איברים, יוצא שבסה"כ ישנם `M*i` איברים בזיכרון לפני האיבר `matrix[i][0]`, ובחשבון מצביעים נקבל:

```
&matrix[i][0] == &matrix[0][0] + M*i
```

# העברת מערך דו ממדי לפונקציה

- משהגענו לאיבר **matrix[i][0]**, נותר לנו רק להתקדם בשורה לאיבר **matrix[i][j]**. לשם כך עלינו לדלג על עוד **j** איברים בזיכרון, ובסיכומו של דבר אנו מקבלים:

```
&matrix[i][j] == &matrix[0][0] + M*i + j
```

**מסקנה**: על מנת לאתר איבר כלשהו במערך דו-ממדי, **הכרחי** לדעת את אורך השורה שלו – דהיינו את מספר הטורים במערך **M** (שימו לב שאת מספר השורות הכולל דווקא אין הכרח לדעת).

זו הסיבה, למשל, מדוע כאשר מאתחלים מערך דו-ממדי חייבים לציין את אורך השורה שלו, גם כאשר מציינים מפורשות רשימת אתחול.

# העברת מערך דו ממדי לפונקציה

- אם כן כיצד נעביר מערך דו ממדי כפרמטר לפונקציה?
- כמו במקרה החד ממדי, נעביר מצביע למערך ולא את תוכנו
- נהיה חייבים לציין באיזשהו אופן מה אורך השורות במערך
- נציין מפורשות את אורך השורה במטריצה באופן הבא:
  נציג פונקציה המקבלת מערך דו ממדי בעל אורך שורה 4.
  חתימת הפונקציה תראה כך:

```
int funky(int matrix[][4]);
```

# העברת מערך דו ממדי לפונקציה

- חשוב לשים לב שהפונקציה שהגדרנו זה עתה יכולה לקבל כפרמטר אך ורק מערכים שאורך השורה שלהם הוא 4! כל ניסיון להעביר לה מערך בעל אורך שורה אחר יגרור שגיאת קומפילציה.

```
int funky(int matrix[][4]);
```

# דוגמא- חישוב מינימום ומקסימום

- נכתוב פונקציה המקבלת מערך דו ממדי כפרמטר ומחזירה את האיבר המינימלי והמקסימלי בו.
- מה תהיה החתימה של הפונקציה?

```
void extremes(int a[][M], int n, int *min, int *max)
```

- שימו לב שבחתימה זו אנו חייבים לציין את אורך השורה של המערך הדו-ממדי (במקרה שלנו זהו **M**, המוגדר כ-**#define**). עם זאת, איננו מניחים דבר בנוגע למספר השורות במערך, ולכן ערך זה מועבר כפרמטר **n** של הפונקציה.

# דוגמא- חישוב מינימום ומקסימום

```
void extremes(int a[][M], int n, int *min, int *max)
{
  int i, j;
  *min = *max = a[0][0];

  for (i = 0; i < n; ++i)
  {
    for (j = 0; j < M; ++j) {
      *min = (a[i][j] < *min) ? a[i][j] : *min;
      *max = (a[i][j] > *max) ? a[i][j] : *max;
    }
  }
}
```

# פוינטר כללי: void*

- הטיפוס **void*** הוא טיפוס לכל דבר, וניתן להגדיר ממנו משתנים.
- טיפוס זה משמש כאשר אנחנו רוצים לייצג כתובת זיכרון "טהורה", ללא כל מידע לגבי הטיפוס שמאוחסן שם.
- ניתן בשפת C להמיר כל מצביע למצביע מטיפוס **void***. למעשה, אנו נפטרים כך מהמידע לגבי הטיפוס שמאוחסן בזיכרון, ונותרים עם כתובת הזיכרון בלבד.
- לא ניתן להפעיל אופרטור * על משתנה מטיפוס **void***.
- אם ידוע לנו הטיפוס האמתי אליו מצביע משתנה מסוג **void***, ניתן להמיר אותו חזרה לסוג המצביע האמתי באמצעות המרה מפורשת.

# void* - דוגמא לשימוש

```
//n is the size (in bytes) to copy
void memcopy(void* dest, void* src,
      int n)
{
    int i;
    char *src_p, *dst_p;

    src_p = (char*)src;
    dst_p = (char*)dest;

    for (i = 0; i < n; i++) {
        *(dst_p+i) = *(src_p+i);
}
```

- פונקציה המעתיקה n בתים של זיכרון.
- חובה להמיר void* לסוג אחר כדי לבצע את ההעתקה. בחרנו char* כיוון שגודלו בית אחד.
- שימוש:

```
int a, b;
memcopy(&a, &b, sizeof(int));//a=b
```