פרק 10
מצביעים ומערכים

מבוא: המערך כטיפוס

- נתבונן בהגדרה הבאה:

```
int a[5], b[6];
```

- נשאלת השאלה, מהו הטיפוס של **a** ו-**b**? היינו מצפים, אולי, כי התשובה תהיה 'מערך של **int**', כלומר משהו כמו **int[]**. אך יש כאן בעיה! לכל טיפוס הרי חייבת להיות מוגדרת כמות הזיכרון שהוא תופס, ואילו ה"טיפוס" **int[]** איננו מוגדר מבחינת כמות הזיכרון שלו.
- יוצא מכך, שלכל אורך של מערך חייב להיות מוגדר טיפוס משלו! בדוגמה למעלה, הטיפוס של **a** הוא למעשה **int[5]**, בעוד הטיפוס של **b** הוא **int[6]**. טיפוסים אלו **שונים**, ולא ניתן להמיר ביניהם.

מבוא: המערך כטיפוס

- כדי להיווכח בהתנהגות זו, ניעזר באופרטור **sizeof**. כמו לכל טיפוס, גם על מערך ניתן להפעיל אופרטור זה, והוא מחזיר (כפי שהיינו מצפים) את סך הזיכרון שהמערך תופס. עבור מערך מטיפוס **T[N]**, יוחזר לפיכך **N·sizeof(T)**.
- ב- Dev-Cpp למשל, השורות הבאות ידפיסו 20:

```
int array[5];
printf("%d\n", sizeof(array));
Printf("%d\n", sizeof(int[5]));
```

- זה המקום לחזור ולהדגיש כי למרות שהתוכנית יודעת את גודלו של המערך, היא עדיין אינה מבצעת כל בדיקת גבולות לאינדקס של האופרטור **[]**. סיבה אחת לכך היא חיסכון בפעולות מעבד, אך יש גם סיבות נוספות שאותן נראה בהמשך.

מערכים ו-typedef

- מקרה נוסף שבו רואים את ההתנהגות הזו הוא כאשר משתמשים ב-**typedef** על מנת להגדיר שם נרדף לטיפוס של מערך. כיוון שכל גודל של מערך נחשב כטיפוס נפרד, עלינו לציין בהצהרת ה-**typedef** את גודלו המדויק של המערך.
- לדוגמה, נגדיר טיפוס בשם **point3d** שייצג נקודה במרחב התלת-מימדי. כל נקודה כזו מאופיינת על ידי שלוש קואורדינאטות ממשיות (x,y,z), ולכן נרצה שהטיפוס **point3d** יהיה שם נרדף לטיפוס **double[3]**.
- נזכיר את הכלל: רושמים את פקודת ה-**typedef** בדיוק כמו שורת הצהרה על משתנה, בתוספת המילה **typedef** בהתחלה:

```
typedef double point3d[3];
```

מערכים ו-typedef

- ניתן כעת להגדיר נקודות במרחב התלת מימדי כך:

```
point3d p1, p2;
```

- שימו לב שהטיפוס האמיתי של שני המשתנים **p1** ו-**p2** הוא למעשה **double[3]**. בהנחה ש-**double** תופש 8 בתים, שתי הפקודות הבאות ידפיסו 24:

```
printf("%d\n", sizeof(point3d));
printf("%d\n", sizeof(p1));
```


מערכים ופונקציות?

- לאור מה שראינו עד כה, מתעוררת בעיה!
- נניח שאנו רוצים לכתוב פונקציה המקבלת מערך כפרמטר. נזכיר כי לכל פרמטר של הפונקציה יש להגדיר במדויק את הטיפוס שלו.
- לפיכך, כיוון שמערכים בעלי אורך שונה הינם מטיפוס שונה, יוצא שנוכל לכתוב את הפונקציה **עבור אורך מערך מסוים בלבד!**
- ברור שלרוב השימושים התנהגות זו אינה נוחה... האומנם נצטרך לממש את הפונקציה שלנו לכל אורך מערך אפשרי?
- בפרק זה נראה שניתן לכתוב פונקציה המקבלת מערך באורך כללי בלא להניח דווקא מאורך מסוים. הכלי העיקרי שנשתמש בו נקרא **חשבון מצביעים**.

אריתמטיקה של מצביעים

- אריתמטיקת מצביעים הינה יכולת חשובה מאוד של שפת C, ונחשבת אחת מהתכונות החזקות של השפה.
- אריתמטיקת מצביעים מאפשרת לקדם מצביעים, וכן לחבר אליהם מספרים שלמים. ניתן לעשות שימוש בכל האופרטורים החיבוריים:

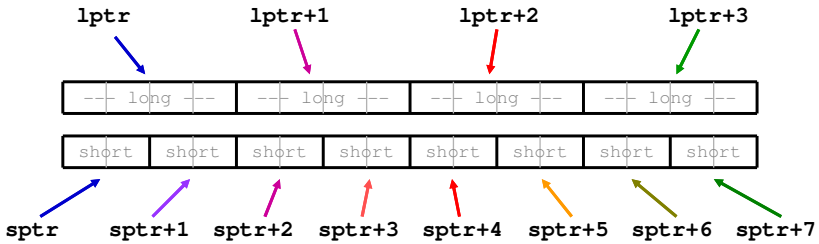
+ - += -= ++ --

- האריתמטיקה מוגדרת כך: כל תוספת/הפחתה של 1 משנה את הכתובת שרשומה במצביע במספר בתים **כגודל הטיפוס אליו מצביעים**.
- לדוגמה: עבור מצביע מטיפוס **double***, תוספת של 1 תקדם אותו **8 בתים** בזיכרון (בהנחה שזהו הגודל של **double** במערכת). לעומת זאת עבור **long***, תוספת של 1 תקדם אותו **ב-4 בתים** בלבד; הפחתה של 3 ממצביע זה תזיז אותו 12 בתים אחורה בזיכרון.

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;
```



דוגמאות לפעולות על מצביעים

```
int *cool_ptr;
```

- קידום **cool_ptr** שלושה **int**-ים קדימה בזיכרון:

```
cool_ptr = cool_ptr + 3;
```

- הזזת **cool_ptr** שני **int**-ים לאחור בזיכרון:

```
cool_ptr = cool_ptr – 2;
```

- צורות נוספות לכתיבת אותן פקודות:

```
cool_ptr += 3;
cool_ptr -= 2;
```


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**.
- לדוגמה, נניח נתון לנו המערך הבא:

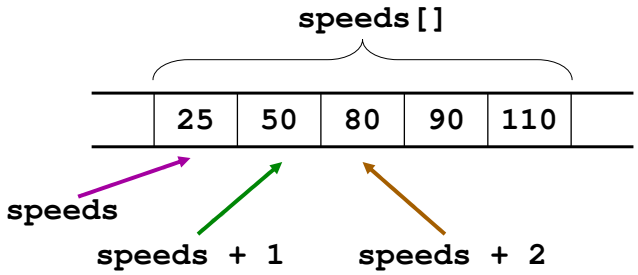
```
int speeds[] = { 25, 50, 80, 90, 110 };
```

- על מנת לקבל את כתובתו של האיבר הראשון במערך, שתי האפשרויות הבאות ייתנו אותה תוצאה:

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


מערכים כמצביעים

- נזכיר שאברי המערך **speeds** נמצאים בזיכרון בצורה רציפה.
- לכן, מרגע שיש לנו מצביע לתחילת המערך, אנו יכולים לקבל מיידית מצביעים לכל איברי המערך באמצעות **אריתמטיקת מצביעים**:



- שימו לב שכל תוספת של 1 למצביע מקדמת אותו מקום אחד במערך.

מערכים כמצביעים

- כלומר, שתי הצורות הבאות שקולות, ומחזירות מצביעים לאברי המערך:

```
speeds + 0
speeds + 1
…
speeds + 4
```

⟷

```
&speeds[0]
&speeds[1]
…
&speeds[4]
```

- יתירה מכך, באמצעות האופרטור * אף ניתן לגשת לאברי המערך עצמם:

```
*(speeds + 0)
*(speeds + 1)
…
*(speeds + 4)
```

⟷

```
speeds[0]
speeds[1]
…
speeds[4]
```

- קיבלנו קשר חשוב בין האופרטורים:

```
x[k]  ↔  *(x + k)
```

מערכים כמצביעים: דוגמאות

- הדפסת תוכן המערך **speeds**:

```
for (i=0; i<5; ++i)
  printf( "%d\n", *(speeds+i) );
```

- בדוגמה הבאה, שימו לב לפרמטר שנשלח ל- **scanf()** (זכרו שפונקציה זו מקבלת מצביע עם הכתובת שאליה יש לכתוב את הקלט!):

```
for (i=0; i<N; ++i) {
  printf("what salary were you thinking of? ");
  scanf("%d", salaries + i);
  *(salaries + i) /= 2;
}
```

&salaries[i]

מצביעים כמערכים

- נגדיר עתה מצביע נוסף לתחילת המערך **speeds**, כך:

```
int *ptr = speeds;
```

- כעת ניתן להשתמש במצביע **ptr** על מנת לגשת לאברי המערך **speeds**, באמצעות אריתמטיקת מצביעים. אברי המערך **speeds** יתקבלו על ידי:

```
*(ptr + 0)
*(ptr + 1)
…
*(ptr + 4)
```

מצביעים כמערכים

- הגיע הזמן לקחת את הקשר בין מערכים ומצביעים לשלב הבא!
- אנו רואים כי ניתן באמצעות המצביע **ptr** לגשת לכל איבר במערך **speeds**, תוך שימוש באריתמטיקת מצביעים. אך האם בזאת מסתיים העניין? מה לגבי האופרטור **[]** ?
- כזכור, ציינו קודם את הקשר הבא כאשר **x** הוא מערך:

```
x[k]    ↔    *(x + k)
```

- למעשה, קשר זה תקף **גם לגבי מצביעים**! כלומר, עבור המצביע **ptr**, הביטוי **ptr[k]** הוא חוקי, ושקול לביטוי ***(ptr+k)** שכבר מוכר לנו.

מצביעים כמערכים

- אנו מקבלים שאם **ptr** מצביע לאיבר הראשון במערך כלשהו, אזי **ptr[k]** יחזיר לנו את האיבר ה-**k** במערך, בדיוק כאילו הפעלנו את האופרטור על שם המערך עצמו.
- כלומר, המצביע **ptr** יכול בעצם לתפקד **כשם נוסף למערך**.
- במקרה שלנו, נוכל לגשת באמצעות המצביע **ptr** לאברי המערך **speeds**, כך:

```
*(ptr + 0)
*(ptr + 1)
…
*(ptr + 4)
```

```
ptr[0]
ptr[1]
…
ptr[4]
```

מצביעים כמערכים

- ועכשיו סוד כמוס: האופרטור **`[]`** מוגדר **אך ורק למצביעים**!
- למעשה, גם כאשר מפעילים את האופרטור **`[]`** על שם של מערך, מה שקורה בפועל הוא ששם המערך הופך למצביע לאיבר הראשון, ועל מצביע זה מופעל האופרטור **`[]`**.
- למשל, כשאנו כותבים **`speeds[2]`** בקוד, השם **`speeds`** ראשית הופך למצביע לתחילת המערך, ואז הפעלת האופרטור **`[]`** עליו מחשבת את **`*(speeds+2)`**. פעולה זו מחזירה את האיבר 2 במערך בדיוק כפי שנעשה עבור מצביע אחר.
- כעת אנו מבינים גם מדוע אין בדיקת טווחים כאשר אנו ניגשים למערך: כיוון שהמערך ראשית הופך למצביע, הרי שהמידע לגבי גודלו אובד בהמרה למצביע, ואז אין עוד אפשרות לוודא את חוקיות האינדקס.
- חישבו: לאור כל זאת, מה באמת מבצעת הפעולה **`speeds[-2]`**?

השוואה: מצביע לעומת מערך

- קיבלנו שמערכים ומצביעים מתנהגים דומה מאוד: כל מערך יכול לתפקד גם כמצביע, וכל מצביע יכול להתחפש למערך.
- ובכן, האם יש הבדלים מהותיים בין מצביע למערך?

- בהחלט כן!
  הבה נראה כמה מהם.

השוואה: מצביע לעומת מערך

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

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

שגיאת קומפילציה

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

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

שגיאת זמן ריצה
(בהנחה שכתובת הזבל ש-**`ptr`** מכיל אינה חוקית)

השוואה: מצביע לעומת מערך

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-Dev-Cpp נקבל:

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

20 4

העברת מערך כפרמטר לפונקציה

- בשפת C מערך איננו יכול להיות פרמטר לפונקציה.
- עם זאת, כיוון ששם המערך מתפקד גם מצביע לאיבר הראשון בו, הדרך להעביר מערך לפונקציה היא על ידי העברת הכתובת שלו.
- מעשית, צורת העברה זו טובה יותר! זאת כיוון שמערך עשוי לתפוש זיכרון רב, ועדיף להמנע משכפול תוכנו בכל פעם שמעבירים אותו כפרמטר לפונקציה.

העברת מערך כפרמטר לפונקציה

העברת מערך לפונקציה נעשית כך:

1. הפונקציה עצמה תקבל **מצביע** בתור הפרמטר שלה:

```
void print(int *array) { … }
```

2. בקריאה לפונקציה, נציין את שם המערך בתור הפרמטר. כרגיל, שם המערך יהפוך אוטומטית למצביע לתחילת המערך, ולכן הטיפוס שלו יתאים לזה שהפונקציה אמורה לקבל. הפונקציה עצמה תקבל את הכתובת של תחילת המערך בזמן הקריאה לה:

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


העברת מערך כפרמטר לפונקציה

3. בתוך הפונקציה, נוכל לעבוד עם המצביע בדיוק כמו עם המערך המקורי! ...בהבדל אחד: כיוון שקיבלנו מצביע ולא את המערך עצמו, איננו יכולים לדעת את אורך המערך שקיבלנו. לכן יש להעביר לפונקציה גם את אורך המערך, כפרמטר נפרד:

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

```
print(my_array, 8);
```


העברת מערך כפרמטר לפונקציה

- נשים לב שלמעשה העברנו את המערך לפונקציה **by reference**. כלומר, הפונקציה לא קיבלה עותק זמני של המערך (שזו העברה by value), 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)
{
  int i;
  for (i=0; i<n; ++i) {
    grades[i] = grades[i] + 5;
  }
}
```

העברת מערך כפרמטר לפונקציה

- הפונקציה הבאה מחשבת ציון ממוצע:

```
double average(int *grades, int n)
{
  int i; double sum=0;
  for (i=0; i<n; ++i) {
    sum += grades[i];
  }
  return sum/n;
}
```

- הקריאה לפונקציות הללו יראו כך:

```
factor(grades, 5);
avg = average(grades, 5);
```


החתימה של פונקציה שמקבלת מערך

- שפת C מאפשרת לכתוב את חתימת הפונקציה הקודמת גם כך:

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

- רגע, הרי אמרנו שמערך איננו יכול להיות פרמטר לפונקציה!
- ובכן, זה עדיין נכון.
- צורת הכתיבה החדשה שקולה לחלוטין לכתיבה הקודמת, והפרמטר **`grades`** הוא עדיין מטיפוס **`int*`**. זו 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 רק עבור פרמטרים של פונקציות! לא ניתן להשתמש בה בהצהרות על משתנים, למשל.

החתימה של פונקציה שמקבלת מערך

- יתרה מכך, מותר לנו אפילו לציין **גודל** למערך בחתימת הפונקציה! עושים זאת כשהפונקציה מצפה לקבל מערך בעל אורך מסוים בלבד, ורוצים להדגיש זאת. עם זאת, אופציה זו היא לנוחות בלבד **ואין לה כל משמעות מעשית**: כלומר, הפרמטר הוא עדיין מצביע, ובתוכנית עצמה נוכל להעביר לפונקציה כל מצביע או מערך מהטיפוס המתאים, ללא קשר לאורכם.
- בדוגמה הבאה, הפונקציה **`norm()`** מחשבת את הנורמה (האורך) של וקטור במרחב התלת-ממדי, המיוצג כמערך של שלושה **`double`**. היא מניחה שהמערך **`v`** מכיל תמיד 3 איברים, ולכן אינה צריכה לקבל כפרמטר נפרד את אורכו. אנו מדגישים עובדה זו בחתימת הפונקציה, אך כמובן שבאופן מעשי, החתימה שקולה לחלוטין לחתימה עם מצביע :

```
double norm(double v[3]);
```

```
double norm(double *v);
```


דוגמה: מיזוג מערכים

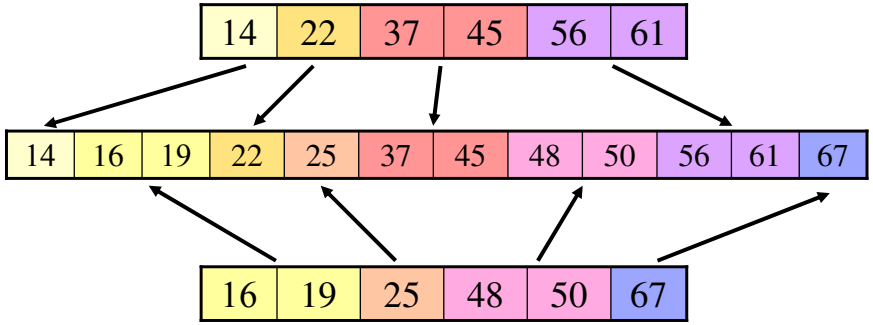
- המצב: נתונים לנו שני מערכים, ממוינים בסדר עולה.
- המטרה: אנו רוצים למזג את שני המערכים למערך אחד גדול, תוך שמירה על סדר נכון של האיברים (כלומר כך שהמערך הגדול ממוין אף הוא בסדר עולה).
- חתימת הפונקציה תהיה:

```
void merge(int a[], int b[], int c[],
           int m, int n);
```

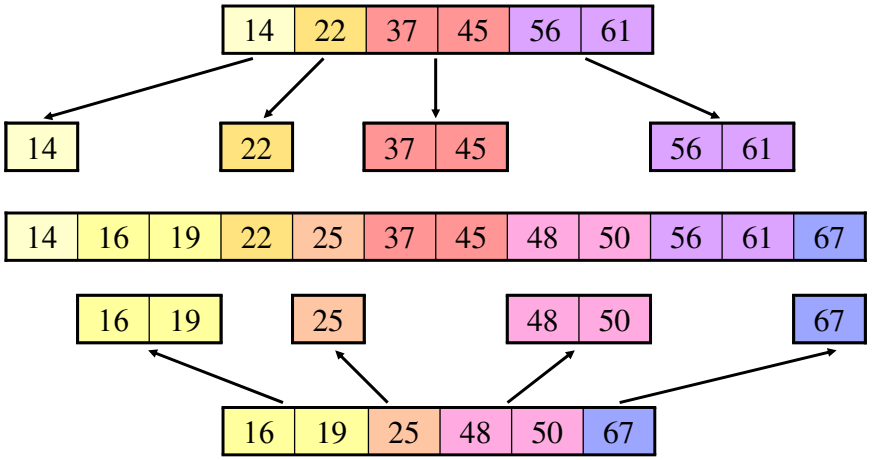
- **`a`** ו-**`b`** הם המערכים למיזוג, ואורכיהם **`m`** ו-**`n`** בהתאמה. היעד אליו יש לכתוב את התוצאה הוא המערך **`c`**. אנו מניחים שבמערך **`c`** יש מספיק מקום לאחסן את התוצאה, כלומר אורכו לפחות ל-**`m+n`**.

האלגוריתם

ניקח רגע לדמיין את פעולת המיזוג. ניתן לראות כי תוצאת המיזוג היא שילוב לסירוגין של "בלוקים" של איברים משני מערכי המקור. כל בלוק כזה הוא רצף של ערכים באחד המערכים, שנמצאים כולם בין שני ערכים עוקבים במערך השני.



האלגוריתם



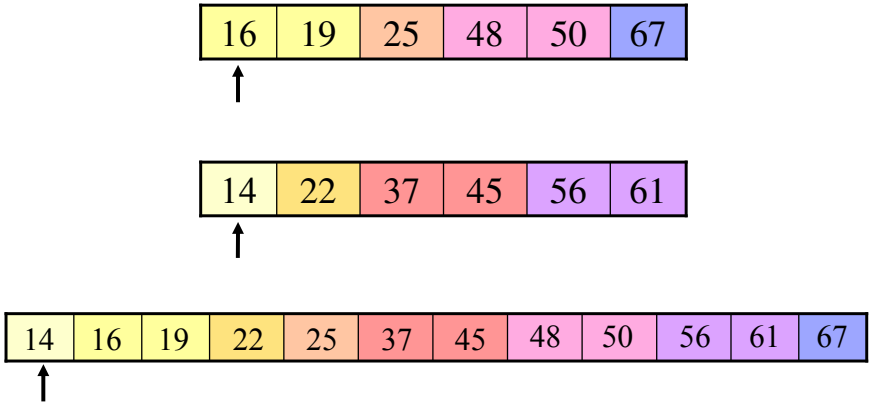
האלגוריתם

כיצד יעבוד האלגוריתם? ובכן, העובדה ששני המערכים ממוינים מאפשרת לנו לבצע את המיזוג **במעבר יחיד** על שני המערכים. אלגוריתם המיזוג יהיה כזה:

- נאתחל שלושה מצביעים, אחד לתחילת כל מערך. מצביעים אלו יכילו את המיקום הנוכחי בכל אחד מהמערכים.
- בכל איטרציה של האלגוריתם, נשווה את שני איברי המקור הנוכחיים, ואת הקטן מביניהם נעתיק למערך היעד. כיוון ששני המערכים ממוינים, ברור שאיבר זה הוא הקטן ביותר מבין כל האיברים שנותרו בשני המערכים. כעת נתקדם מקום אחד קדימה במערך שממנו לקחנו את האיבר (וכן גם במערך היעד), ונמשיך.
- האלגוריתם נמשך כך עד שאחד ממערכי המקור מסתיים. לבסוף, מעתיקים את שארית המערך השני לתוך מערך היעד.

האלגוריתם

דוגמה לריצת האלגוריתם:



הפונקציה למיזוג מערכים (אופציה 1)

```
void merge(int a[], int b[], int c[], int m, int n)
{
  int i=0, j=0, k=0;

  while ((i < m) && (j < n)) {
    c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
  }

  while (i < m)
    c[k++] = a[i++];

  while (j < n)
    c[k++] = b[j++];
}
```


השוואה בין מצביעים

- נראה כעת דרך חילופית לכתוב את הקוד של פונקצית המיזוג, ובה לא נשתמש באופרטור `[]`, אלא נקדם את המצביעים עצמם.
- לשם כך, עלינו להכיר פעולה נוספת שניתן לבצע על מצביעים – השוואה. 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:

`>` `<` `>=` `<=` `==` `!=`

הפונקציה למיזוג מערכים (אופציה 2)

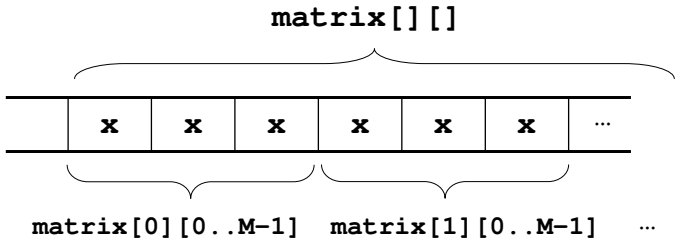
```
void merge(int a[], int b[], int c[], int m, int n)
{
  int *a_end = a+m, *b_end = b+n;

  while ((a < a_end) && (b < b_end)) {
    *(c++) = (*a < *b) ? *(a++) : *(b++);
  }

  while (a < a_end)
    *(c++) = *(a++);

  while (b < b_end)
    *(c++) = *(b++);
}
```


העברת מערך דו-ממדי לפונקציה

- נתבונן במערך הדו-ממדי הבא:

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

- ניזכר כי מערך זה פרוש בזיכרון שורה-אחר-שורה, כך:



העברת מערך דו-ממדי לפונקציה

באמצעות חשבון מצביעים, אנו יכולים כעת לחשב מצביע לאיבר **matrix[i][j]** בזיכרון. נעשה זאת בשלבים:

1. כתובת תחילת המערך הדו-ממדי היא:

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

2. עתה, נחשב מצביע ל-**matrix[i][0]** – האיבר הראשון בשורה **i**. 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

3. 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
```

מטרת החישוב הזה הינה המסקנה החשובה הבאה:

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

לסיום, נדון בקצרה באופן בו מעבירים מערך דו-ממדי כפרמטר לפונקציה.

- העברת מערך דו-ממדי לפונקציה נעשית באופן דומה למקרה החד-ממדי, כלומר מה שמועבר למעשה הוא מצביע למערך ולא התוכן שלו.
- כפי שראינו, ניתן לאתר איבר במערך דו-ממדי רק בהינתן אורך השורה שלו; מסקנה זו תקפה כמובן גם עבור פונקציה שמקבלת מערך דו-ממדי – שהרי ללא מידע זה, כל מה שיהיה בידיה הוא רק כתובת האיבר הראשון, מה שאינו מספיק לצורך איתור יתר האיברים. לכן, בחתימת הפונקציה יש לציין מפורשות את אורך השורה של המטריצה.
- לדוגמה, הפונקציה הבאה מקבלת 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.
- בדומה למקרה החד-ממדי, במידה והפונקציה מניחה שהמערך מכיל מספר מסוים של שורות, אזי ניתן בחתימת הפונקציה לציין גם את מספר השורות במערך הדו-ממדי. עם זאת, ציון מספר השורות במערך הוא חסר משמעות מבחינת הקומפיילר, והוא מתעלם מכך לחלוטין:

```
int funky(int matrix[7][4])
```

- כמו כן, בדומה למקרה החד-ממדי, גם כאן לחתימות אלו קיימת גרסה שקולה המכילה מצביע מפורש כפרמטר. עם זאת, הטיפוס של מצביע זה הוא סבוך יותר, ולא ניגע בכך בקורס זה.

דוגמא – חישוב מינימום ומקסימום

- נביא כדוגמה פונקציה פשוטה שמקבלת מערך דו-ממדי כפרמטר, ומחזירה את האיבר המינימאלי והאיבר המקסימאלי בו.
- כיוון שעלינו להחזיר שני ערכים, נעשה זאת על ידי העברת שני מצביעים לפונקציה, שלתוכם היא תכתוב את התוצאות.
- חתימת הפונקציה תהיה:

```
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;
    }
  }
}
```