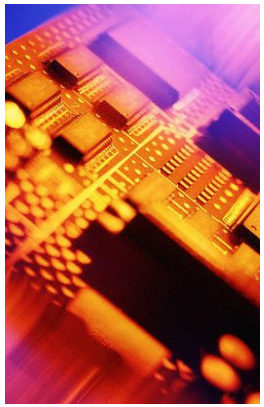
פרק 9
מצביעים

כתובות זיכרון

- הזיכרון במחשב מחולק לתאים, כל אחד בגודל בית אחד. תאי זיכרון אלה ממוספרים, ומספרי התאים נקראים גם **הכתובות** של התאים.
- לכל תוכנית מחשב מוקצה אזור עבודה בזיכרון. אוסף כתובות הזיכרון בהן התוכנית יכולה לאחסן נתונים נקרא **מרחב הכתובות** של התוכנית.
- כל משתנה מאוחסן באחד או יותר תאים רצופים בזיכרון. **כתובת המשתנה** היא כתובתו של התא הראשון המכיל אותו.
- כאשר התוכנית פונה לזיכרון, עליה לציין את הכתובת הרצויה ואת מספר הבתים שתופש המשתנה.



האופרטור &

- ניתן לקבל את כתובתו של משתנה כלשהו **var_name** כך:

```
&var_name
```

- ניתן להדפיס כתובות עם **printf()** באמצעות עם התג **%p**. כתובות מודפסות כמספר בבסיס הקסדצימאלי (בסיס 16).

```
int main() {
  int x;
  double d;

  printf("address of x: %p\n", &x);
  printf("address of d: %p\n", &d);
  return 0;
}
```

```
address of x: 0022FF74
address of d: 0022FF68
```


מצביעים (pointers)

- שאלה: כל ביטוי ב-C חייב להיות בעל טיפוס מסוים. מה הטיפוס שמחזיר האופרטור **&** ?
- תשובה: תלוי בביטוי שעליו מפעילים את ה-**&** ! לכל טיפוס **T** ב-C קיים **טיפוס מיוחד**, שנקרא **T***, לייצוג כתובות של משתנים מטיפוס **T**.
- הטיפוס **T*** הוא טיפוס לכל דבר, שניתן להגדיר ממנו משתנים. משתנים אלו נקראים **מצביעים**, והם מכילים כתובות זיכרון. למשל, משתנה מטיפוס **int*** (מצביע ל-**int**) מכיל כתובות של משתנים מטיפוס **int**.

מצביעים (pointers)

- מצביע, כמו כל משתנה, צריך להיות מאוחסן במקום כלשהו בזיכרון, כיוון שהוא צריך לאחסן מידע (במקרה זה כתובת).
- אפשר לבדוק היכן המצביע עצמו מאוחסן – בעזרת האופרטור **&** כמובן!
- איזה טיפוס יחזיר האופרטור **&** הפעם? ובכן לפי הכלל, אם המצביע שלנו הוא מטיפוס **int***, הרי שהטיפוס שהאופרטור **&** יחזיר הוא **int****.

```
int x;
int*  ptr1 = &x;
int** ptr2 = &ptr1;

printf("address of x:    %p\n",  ptr1);
printf("address of ptr1: %p\n",  ptr2);
printf("address of ptr2: %p\n", &ptr2);
```

```
address of x:    0022FF74
address of ptr1: 0022FF70
address of ptr2: 0022FF6C
```

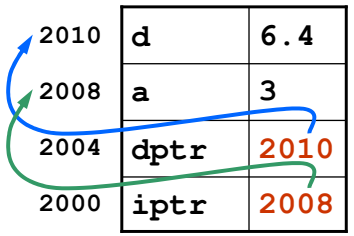

דוגמאות למצביעים

- נגדיר כמה מצביעים. שימו לב שהמצביעים הם משתנים לכל דבר, ולכן הם מאוחסנים במחסנית כמו כל משתנה אחר.
- כמות הבתים הדרושה לאחסון כתובת זיכרון תלויה בארכיטקטורה של המחשב עליו רצה התוכנית, אך **היא זהה לכל סוגי המצביעים** ללא תלות בטיפוס עליו מצביעים. ברוב המחשבים כיום מצביע תופס 4 בתים.

```
int* iptr;
double* dptr;

int a = 3;
double d = 6.4;

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



האופרטור * (כוכבית)

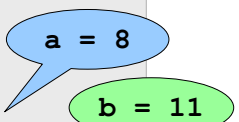
- בהינתן מצביע, ניתן לעבוד ישירות עם המשתנה שעליו הוא מצביע, באמצעות האופרטור * (כוכבית):

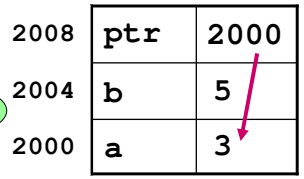
```
*ptr_name
```

- נניח **ptr** הוא מצביע מטיפוס ***T**, כלומר הוא מצביע למקום בזיכרון שבו מאוחסן משתנה מטיפוס **T**. באמצעות האופרטור * , ניתן לעבוד ישירות אם משתנה זה – כלומר ***ptr** מתנהג ממש כמו משתנה רגיל מטיפוס **T**.

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

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





הצהרה על מצביעים

- לשם הגדרת מצביע ל-**int**, כל צורות הכתיבה הבאות יעבדו:

```
int*  ptr;
int  *ptr;
int * ptr;
```

- אולם, נניח שאנו רוצים יותר ממצביע אחד. ננסה להגדיר כך:

```
int* ptr1, ptr2;
```

- צורת כתיבה זו לא תגרום לתוצאה שרצינו! הבה נראה למה.

הצהרה על מצביעים

- נתבונן שוב בהצהרה:

```
int* ptr1, ptr2;
```

- למעשה, בהצהרה זו ה-* מתייחסת **אך ורק** ל-**ptr1**, ורק הוא יוגדר להיות מטיפוס **int***. לעומת זאת, **ptr2** יוגדר להיות מטיפוס **int**, ולא כפי שרצינו.
- על מנת למנוע בלבול, יש לכתוב את השורה הקודמת כך:

```
int *ptr1, ptr2;
```

כעת ברור כי רק **ptr1** הוא מצביע ואילו **ptr2** הוא מספר שלם.

הצהרה על מצביעים

- הדרך הנכונה להסתכל על ההצהרה

```
int *ptr1, ptr2;
```

היא שהצהרנו על כך ש-**ptr2** ו-***ptr1** הינם מטיפוס **int**. שימו לב שבשורה זו הקצנו זיכרון ל-**ptr2** לאחסון מספר שלם, ואילו ל-**ptr1** לאחסון כתובת (בלבד) של מספר שלם.

- שוב, אין כאן הקצאת מקום כלשהי ל-***ptr1** ! כלומר, לא ניתן באמצעות **ptr1** לאחסן מספר שלם. יתירה מכך, כיוון ש-**ptr1** לא אותחל, הרי שהוא מכיל כרגע כתובת זבל שכלל לא בטוח שאפילו קיימת.
- לסיכום, אילו היינו רוצים שני מצביעים, הדרך לעשות זאת הייתה כך:

```
int *ptr1, *ptr2;
```


מה אפשר לעשות עם מצביעים?

- חשוב להבחין שמצביע בשפת C איננו מייצג רק כתובת בזיכרון, אלא הוא כולל גם אינפורמציה לגבי **סוג המידע** שנמצא בכתובת זו.
- בזכות תכונה זו, הטיפוס של ***ptr** ידוע ומוגדר היטב, וניתן לעבוד איתו כמשתנה לכל דבר.
- כלומר, מרגע שיש בידינו מצביע למקום כלשהו בזיכרון, אנו יכולים באמצעות האופרטור * לקבל את הנתון שנמצא שם, ואף לשנות אותו.
- חשיבות המצביעים היא בכך שמרגע שנתונה לנו כתובת זיכרון, אנו מקבלים **גישה ישיר לזיכרון הזה**, ואין עוד משמעות לשאלות כמו היכן הוא הוקצה או על ידי מי. אנו משוחררים ממגבלות כמו טווחי הכרה של משתנים, הסתרת שמות של משתנים וכל אלו.

מצביעים ופונקציות

- כיוון שמצביע הוא משתנה לכל דבר, ניתן להעביר אותו כפרמטר לפונקציה, ובכך לאפשר לה לעבוד ישירות עם הזיכרון אליו הוא מצביע.
- שימו לב שהעברת המצביע לפונקציה נעשית (כמו תמיד) by value, כלומר ערך המצביע מועתק למשתנה פנימי של הפונקציה.
- אילו יתרונות יש להעברת מצביע לפונקציה?

- בשפת C ניתן להגדיר טיפוסים התופסים זיכרון רב (למשל מערכים). במקרה זה, העברתם לפונקציה by value (כלומר על ידי העתקה) תהיה פעולה כבדה מאוד. ניתן לחסוך את עבודת ההעתקה הזו על ידי העברת **הכתובת** של המידע לפונקציה במקום את תוכנו.
- העברת הכתובת של משתנה לפונקציה מאפשרת לה **לשנות את ערכו של המשתנה**, כיוון שאנו מקנים לה גישה ישירה לזיכרון שלו.

דוגמאות לפונקציות עם מצביעים

- הפונקציה הבאה מבצעת השמה למשתנה. היא מקבלת כתובת, ורושמת בה את הערך הנתון:

```
void assign(int* ptr, int val) {
  *ptr = val;
}
```

- על מנת להשתמש בפונקציה, שימו לב שיש להעביר לה את הכתובת של המשתנה אליו אנו רוצים לשים ערך:

```
int x;
assign(&x , 5);
```


דוגמאות לפונקציות עם מצביעים

- פונקציה שמחליפה את התוכן של שני ממשיים:

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

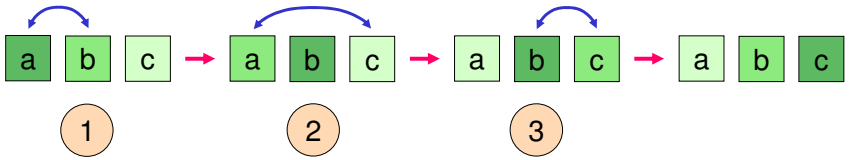
- ניתן להשתמש בפונקציה זו להחלפת תוכנם של זוג משתנים, כך:

```
float a = 3, b = 7;
swap(&a, &b);
printf("a=%.2f, b=%.2f", a, b);
```

```
a=7.00, b=3.00
```


דוגמאות לפונקציות עם מצביעים

- נכתוב פונקציה שמקבלת שלושה מצביעים, a, b ו-c, למספרים שלמים, ומחליפה את תוכנם כך שיהיו מסודרים בסדר עולה.
- האלגוריתם: ראשית נסדר את a ו-b כך שיהיו בסדר עולה. כעת a מכיל את המספר הקטן מבין השניים. נשווה את ערכו ל-c, ובמידת הצורך נחליף ביניהם. בתום שתי פעולות אלו a מכיל את המספר הקטן מבין כל השלושה. לסיום, נסדר את b ו-c כך שיהיו בסדר עולה גם כן.



דוגמאות לפונקציות עם מצביעים

- נעזר בפונקציה **swap()** לצורך ביצוע ההחלפות. שימו לב ש-**a**, **b**, ו-**c** הם כבר מצביעים ולכן אין צורך להפעיל עליהם את האופרטור **&** בקריאה אליה:

```
void sort3(float* a, float* b, float* c)
{
  if (*a > *b) swap(a,b);
  if (*a > *c) swap(a,c);
  if (*b > *c) swap(b,c);
}
```

- פונקציה נוספת שכבר העברנו לה כתובות היא **scanf()**! כזכור, פונקציה זו צריכה לשנות את ערכם של משתנים. לפיכך, אנו נותנים לה את הכתובות שלהם, וכך היא יכולה לרשום לתוכם את הקלט.

המרת מצביעים

- מה יקרה אם ננסה באמצעות הפונקציה **swap()** להחליף את ערכיהם של שני משתנים מטיפוס **double**? אנו יודעים שניתן להמיר אוטומטית **double** ל-**float**, ולכן ננסה להשתמש בפונקציה שכתבנו:

```
double a = -2, b = 2;
swap(&a, &b);
printf("a=%lf , b=%lf\n", a, b);
```

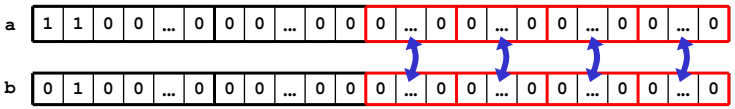
- הרצת התוכנית ב-Dev-Cpp נותנת:

```
a = -2 , b = 2
```

- רגע... המשתנים כלל לא הוחלפו! התוכנית לא ביצעה את מה שרצינו!

המרת מצביעים

- הבעיה נובעת מכך **שכל טיפוס מאוחסן אחרת בזיכרון**. לכן, אם ננסה לפנות לזיכרון שיש בו **double** מתוך מחשבה שיש שם **float**, התוכנית שלנו תתנהג בצורה בלתי צפויה.
- השרטוט הבא מדגים את הבעיה: הפונקציה **swap()** מתייחסת לכתובות שקיבלה ככתובות של **float**. לכן, כשהיא מבצעת את ההחלפה היא מחליפה רק ארבעה בתים, בעוד ש-**a** ו-**b** הם באורך שמונה בתים כל אחד:



- מסקנה: ב-C יש תמיד לדאוג **להתאמה מלאה** בין טיפוסי מצביעים.

הטיפוס void*

- הטיפוס **void*** הוא טיפוס לכל דבר, וניתן להגדיר ממנו משתנים.
- טיפוס זה משמש כאשר אנו 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 האמיתי אליו מצביע משתנה מסוג **void***, ניתן להמיר אותו חזרה לסוג המצביע האמיתי באמצעות המרה מפורשת.

דוגמאות ל- void*

נרחיב את הפונקציה **swap()** כך שתתמוך במגוון טיפוסים. אנו נסתמך על פרמטר נוסף בכדי לדעת את הטיפוס האמיתי של **p** ו-**q**:

```
#define INT 0
#define SHORT 1
#define DOUBLE 2
   ⋮
void swap(void* p, void* q, int type) {

  switch (type) {
    case (INT): {
      int temp   = *((int*)p);
      *((int*)p) = *((int*)q);
      *((int*)q) = temp;
      break;
    }
    case (DOUBLE): {
      double temp = …
```

```
int a, b;
swap(&a, &b, INT);
```

דוגמאות ל- void*

הערות לגבי הפונקציה **swap()** :

- לביצוע ההחלפה אנו זקוקים למשתנה **temp**. אולם כיוון שאיננו יודעים מראש את הטיפוס שמוצבע ע"י **p** ו-**q**, הרי שאי אפשר להצהיר על **temp** מראש. כדי לפתור זאת, **temp** מוקצה בתוך ה-**case**, כאשר הטיפוס הדרוש כבר ידוע. שימו לב שאנו פותחים בלוק חדש עם סוגריים מסולסלים **{}**, מה שמאפשר לנו להצהיר על המשתנה החדש.
- בתוך ה-**case**, כאשר הטיפוס של **p** ו-**q** כבר ידוע, ניתן לבצע המרה מפורשת של **p** ו-**q** לטיפוסם האמיתי. כך למשל, הביטוי **(int*)p** ממיר את **p** למצביע ל-**int**, ולכן ניתן להפעיל עליו את האופרטור ***** ולקבל גישה ל-**int** שעליו הוא מצביע. לפיכך הביטוי ***(int*)p** נותן לנו גישה לתוכן ש-**p** מצביע עליו, תחת ההנחה שהוא מצביע ל-**int**.

דוגמאות ל- void*

- נכתוב כעת פונקציה שמעתיקה בלוק זיכרון של **n** בתים מכתובת מקור לכתובת יעד. שתי הכתובות יהיו נתונות כ-**void***.
- על מנת להעתיק את הבלוק במלואו, נבצע לולאה באורך **n**, שבה בכל איטרציה נעתיק בית אחד מהמקור ליעד.
- בהינתן כתובת זיכרון שמאוחסנת במצביע **void***, היינו רוצים גישה ישירה לבית שנמצא בכתובת זו. ניתן לעשות זאת על ידי המרת המצביע לטיפוס **char***, והפעלת האופרטור ***** על התוצאה. שימו לב שאין לנו כל אין עניין בטיפוס האמיתי של המידע שנמצא בכתובת שקיבלנו! אנו פשוט מנצלים את העובדה ש-**char** מאוחסן תמיד בבית אחד, ולכן כשמפעילים על מצביע **char*** את האופרטור *, מקבלים בדיוק את תוכן הבית שנמצא בכתובת זו.

דוגמאות ל- void*

```
void memcopy(void* dest, void* src, size_t n)
{
  size_t i;
  char *src_p, *dst_p;

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

  for (i=0; i<n; ++i) {
    *dst_p = *src_p;
    dst_p = dst_p + 1;
    src_p = src_p + 1;
  }
}
```

הטיפוס שאיתו מקובל לייצג כמויות זיכרון (זה הטיפוס ש-**sizeof** מחזיר)

הוספת 1 למצביע מקדמת אותו בית אחד בזיכרון. נדון בנושא זה בהרחבה בהמשך.

דוגמאות ל- void*

נשתמש בפונקציה שמימשנו על מנת להעתיק תוכן של משתנה למשתנה אחר. שימו לב שבקריאה ל-**memcopy()** שני המצביעים מומרים אוטומטית לטיפוס **void***:

```
int main() {
  int x = 5, y;
  double d = 3.14, e;

  memcopy(&y, &x, sizeof(int));
  memcopy(&e, &d, sizeof(double));

  printf("%d %d\n", x, y);
  printf("%lf %lf\n", d, e);
  return 0;
}
```

```
5 5
3.140000 3.140000
```

הכתובת 0 והקבוע NULL

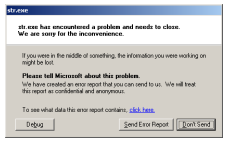
- הכתובת 0 הינה כתובת מיוחדת. זו אינה כתובת חוקית בזיכרון, אלא ערך מיוחד שניתן לשים בכל מצביע.
- הכתובת 0 מיועדת להשמה במצביע כשרוצים לציין שהוא אינו מכיל כל כתובת חוקית. **יש לאתחל ל-0** כל מצביע שאינו מאותחל לזיכרון חוקי כלשהו, וכמו כן יש לשים 0 במצביע במידה והזיכרון שאליו הוא מצביע מפסיק להיות מוקצה.
- במקום הערך 0, ניתן גם להשתמש בקבוע **NULL** – זהו **#define** שמוגדר אוטומטית כאפס.

```
int *p, *q;
p = 0;
q = NULL;
```


הכתובת 0 והקבוע NULL

- מנגנון זה מאפשר לבדוק את חוקיותו של כל מצביע לפני השימוש בו:

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

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

```
int *p = 0;
*p = 3;
```