

נושאי השיעור היום: כתובות ומצביעים

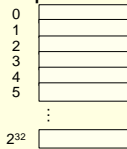
- מהן כתובות בזיכרון
- פעולות עם כתובות
- מהם מצביעים ומה אפשר לעשות איתם
- מצביעים ומערכים

קורס תכנות

שיעור תשיעי: מצביעים

זיכרון - תזכורת

- רצף (מערך) של בתים (bytes)
- כל בית מזוהה על ידי כתובתו
- הכתובת היא האינדקס של הבית במערך
- הכתובת היא מספר
- הכתובת 0 שמורה ואינה מקום חוקי בזיכרון



כתובות בזיכרון

- לכל תא בזיכרון המחשב יש כתובת (מספר)
- משתנה יכול לתפוס כמה תאים בזיכרון (בהתאם לטיפוס המשתנה)
- כתובת המשתנה היא כתובת התא הראשון אותו הוא תופס
- למשל, כשמגדירים משתנה על ידי `int i`
- הקומפילר מקצה עבור המשתנה `i` מקום בזיכרון
- שומר טבלה של כתובות המשתנים

כתובות בזיכרון

```
int main()
{
    int x = 10;
    char c = 'a';

    ...
    return 0;
}
```

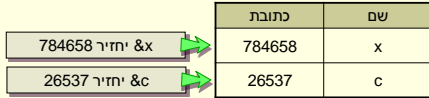
10	'a'
784658	26537
כתובת	שם
784658	x
26537	c

כתובות בזיכרון – גודל המשתנה

- בטבלת הכתובות נשמר נתון נוסף - מספר התאים שמשמשים לייצוג המשתנה שהוגדר
- תלוי בטיפוס המשתנה

כתובות בזכרון

- האופרטור &
- מחזיר את כתובת המשתנה בזיכרון.



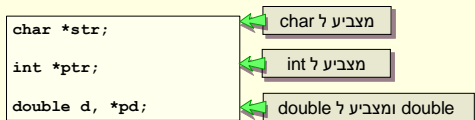
כיצד נשמור כתובות?

- מספרים שלמים נשמור במשתנה מטיפוס int
- מספרים ממשיים נשמור במשתנה מטיפוס double
- וכתובות?
- משתנה שמחזיק כתובת נקרא מצביע (pointer)
- טיפוס המצביע תלוי בטיפוס המשתנה שאליו הוא מצביע

הגדרת מצביע (pointer variable)

`type *variable_name;`

- למשתנה מצביע יש שם
- לפני השם תופיע כוכבית * כדי לסמן שזהו מצביע
- הטיפוס מתאר את המשתנה שמצביעים עליו



למה יש מצביע שונה לכל טיפוס?

- סוג המצביע מציין את:
 - גודל הזכרון של המשתנה שמצביעים עליו
 - סוג התוכן שבכתובת הזאת
- לפי סוג המצביע
 - אנו יודעים בכמה בתים מיוצג המשתנה באותה כתובת
 - אנו יודעים כיצד לפרש את תוכן התאים באותה כתובת

מצביעים וכתובות

```
int main()
{
  int x = 10;
  char c = 'a';

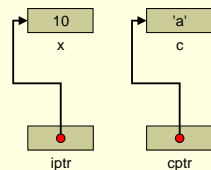
  int *iptr = &x;
  char *cptr = &c;
  ...
  ...
  return 0;
}
```

ערך	כתובת	שם
10	784658	x
'a'	26537	c
784658	43745	iptr
26537	32545	cptr

מצביעים וכתובות

```
int main()
{
  int x = 10;
  char c = 'a';

  int *iptr = &x;
  char *cptr = &c;
  ...
  ...
  return 0;
}
```



האופרטור * (dereference)

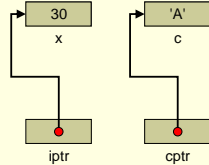
- ניגש לזיכרון הנמצא בכתובת מסוימת

```
int main()
{
    int x = 10;
    char c = 'a';

    int *iptr = &x;
    char *cptr = &c;

    *iptr = 30;
    *cptr = 'A';

    ...
    ...
    return 0;
}
```



הגבלות על פעולות

- לא ניתן לשנות כתובת של משתנה

✗ `&i = 6000;`
`&i = #`

- לקבועים ולביטויים אין כתובת

✗ `i = &5;`
`i = &(6*a+1);`

- אסור לגשת לזיכרון שלא שייך לנו

✗ `*(4000) = 5;`

מצביע ל"כלום"

- NULL מסמל כתובת לא חוקית
- קבוע המוגדר בספרייה `stdlib.h`
- כדי לסמן שמצביע אינו מצביע למשתנה ניתן לו את הערך NULL

```
int *my_pointer = NULL;
```

- נוכל לבדוק אם המצביע מכיל כתובת חוקית

```
if (my_pointer == NULL)
{
    ...
}
```

מצביע כללי

- נוכל להחזיק מצביע (כתובת) ללא ציון הטיפוס המוצבע

```
void *my_void_pointer = NULL;
```

- קיימות מגבלות על השימוש ב `void pointer`
- לא ניתן לגשת למשתנה בכתובת בעזרת האופרטור *
- לא ניתן לבצע פעולות אריתמטיות

- משמש להחזקת כתובת שנידרש להמיר אותה לטיפוס ספציפי כדי להשתמש בה

סיכום ביניים

- משתנה מוגדר ע"י טיפוס ושם (`int k`)
- משתנה מצביע מוגדר ע"י טיפוס ושם (`int *ptr`)
- הכוכבית מסמנת שהמשתנה הוא מצביע והטיפוס מציין את הטיפוס של המשתנה שמצביעים אליו
- אחרי שהגדרנו משתנה נוכל להשיג את כתובתו בעזרת האופרטור &
- ניתן לגשת לערך שהמצביע מצביע עליו בעזרת האופרטור *

הדפסת מצביעים

```
int x = 5;
int *pointer = &x;
printf("address: %p\n", pointer);
```

```
address = 0012FF60
```

- ההדפסה של כתובת תהיה בבסיס הקסדצימלי (בסיס 16)
- בספירה ההקסדצימלית הספרות מ-0 עד 9 נראית כמו דומותיהן העשרוניות, והספרות הבאות הן האותיות מ-A, המייצגת 10, עד F, המייצגת 15.

אריתמטיקה של מצביעים

- ניתן לבצע פעולות אריתמטיות על מצביעים
- כתובות הן מספרים
- קיימים הבדלים בין פעולות על מספרים ובין פעולות על מצביעים

```
char *cptr = (char*) 2;
printf("cptr before: %p\n", cptr);
cptr++;
printf("and after: %p\n", cptr);
```

output:

```
cptr before: 00000002
and after: 00000003
```

אריתמטיקה של מצביעים (המשך)

- נחליף את המצביע מ `char*` ל `int*`

```
int *iptr = (int*) 2;
printf("iptr before: %p\n", iptr);
iptr++;
printf("and after: %p\n", iptr);
```

output:

```
iptr before: 00000002
and after: 00000006
```

- מקדם את המצביע למשתנה הבא
- תלוי בטיפוס המשתנה
- לא ניתן לבצע פעולות אריתמטיות על `void*`

מצביעים ומערכים

- קיים קשר הדוק בין מצביעים למערכים
- בדרך כלל נוכל להתייחס אליהם כאותו הדבר
- שם המערך הוא הכתובת של האיבר הראשון במערך
- מצביע שלא ניתן לשנות את ערכו

מערכים

- כשמגדירים מערך

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

- יוקצה זיכרון רציף המספיק לעשרה משתנים מטיפוס `int`
- `array` הוא הכתובת של התא הראשון במערך
- `array` שקול ל `&array[0]`
- אי אפשר לשנות כתובת של מערך

```
✗ array = (int*) 2;
```

השמה בין מערך למצביע

- מצביע מכיל כתובת, שם המערך הוא כתובת
- ניתן לבצע השמה בין המערך ומצביע

```
int *ptr;
int array[10];

ptr = array;
```

- לאחר ההשמה ניתן להשתמש במצביע כאילו היה שם המערך
- ניתן להשתמש במערך כאילו היה מצביע

```
*array = 4;
```

כיצד זה עובד

- הקומפילר מתרגם גישה לאיבר במערך לגישה לכתובת האיבר המתאים
- הפקודה `array[5] = 100`
- תתורגם ל - `*(array + 5) = 100`

- הפעולות הבאות הן שקולות

```
array[5] = 100;
*(array + 5) = 100;
*(ptr + 5) = 100;
```

שימוש בלולאות

```
int i, array[10], *ptr;
```

- מעבר על איברי המערך בעזרת אינדקס

```
for (i = 0; i < 10; i++) {  
    printf("%d ", array[i]);  
}
```

- גישה לפי כתובת יחסית להתחלה

```
for (i = 0; i < 10; i++) {  
    printf("%d ", *(array + i));  
}
```

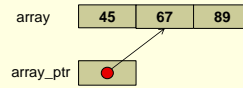
- גישה בעזרת מצביע

```
for (ptr = array; ptr < &array[10]; ptr++) {  
    printf("%d ", *ptr);  
}
```

עוד דוגמא: מה יודפס?

```
int array[] = {45, 67, 89};  
int *array_ptr = &array[1];  
printf("%d\n", array_ptr[1]);
```

89



מצביעים ופונקציות

- ניתן להעביר כתובת (מצביע) כפרמטר לפונקציה
- ניתן לקבל כתובת (מצביע) כערך מוחזר
- לא ניתן לשנות את הערך של המצביע המועבר לפונקציה
 - כמו כל משתנה אחר
- ניתן לשנות את המשתנה שמצביעים עליו

Swap כפונקציה

- ראינו כבר קוד לביצוע פעולת ההחלפה (Swap)

```
int temp, i, j;  
...  
temp = i;  
i = j;  
j = temp;
```

- כיצד נהפוך קטע קוד זה לפונקציה?

```
void swap(int i, int j)  
{  
    int temp;  
    temp = i;  
    i = j;  
    j = temp;  
}
```

?

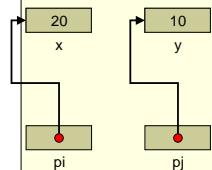
פתרון בעזרת מצביעים

- נעביר לפונקציה את הכתובת של המשתנים במקום את ערכי המשתנים

```
void swap(int *pi, int *pj)  
{  
    int temp;  
    temp = *pi;  
    *pi = *pj;  
    *pj = temp;  
}
```

דוגמא

```
void swap(int *pi, int *pj)  
{  
    int temp = *pi;  
    *pi = *pj;  
    *pj = temp;  
}  
  
int main()  
{  
    int x = 10;  
    int y = 20;  
  
    swap(&x, &y);  
    printf("x is %d - y is %d\n", x, y);  
    return 0;  
}
```



דוגמא שפגשנו כבר

- הפונקציה scanf
- מקבלת את כתובת המשתנה שאליו יוכנס הקלט
- המשתנה grade לא מוגדר בפונקציה scanf
- הפונקציה scanf מקבלת את הכתובת של המשתנה בזיכרון

```
int main()
{
    int grade;
    ...
    scanf("%d", &grade);
    ...
    return 0;
}
```

למה צריך מצביעים?*

- משתנה
- ניתן לשנות את ערכו אך ורק בפונקציה בה הוגדר
- מצביע
- נותן לנו גישה אל משתנים שהוגדרו בפונקציות אחרות
- דוגמא: הפונקציה scanf משנה ערכים של משתנים
- פונקציה יכולה לחשב מספר ערכים ולשמור אותם בכתובות שהועברו אליה (כמו להחזיר מספר ערכים מפונקציה)

*עוד סיבות בהמשך

מצביע כערך מוחזר

```
#include <stdio.h>

int* max(int *a, int *b)
{
    return (*a > *b) ? a : b;
}

int main()
{
    int i = 10, j = 20;
    int *max_ptr = max(&i, &j);

    printf("&i = %p, &j = %p, max_ptr = %p\n", &i, &j, max_ptr);

    return 0;
}
```

```
&i = 0012FF60, &j = 0012FF54, max_ptr = 0012FF54
```

שימו לב!

- אין להחזיר את הכתובת של משתנה לוקאלי
- משתנים לוקאליים אינם קיימים לאחר שהפונקציה הסתיימה

```
int* pointer_to_zero()
{
    int zero = 0;
    return &zero;
}
```



מערך כפרמטר לפונקציה

- כאשר מועבר מערך בפרמטר לפונקציה מועברת הכתובת של תחילת המערך
- לכן ניתן לשנות את איברי המערך
- העברת מערך שקולה להעברת מצביע
- איך נדע להבדיל?
- לא נדע מהתחביר, צריך לקרוא את התיעוד

הגדרות שקולות

```
void init_array(int array[], int size)
{
    int i;

    for (i = 0; i < size; i++)
        array[i] = 0;
}

void init_array(int *array, int size)
{
    int i;

    for (i = 0; i < size; i++)
        array[i] = 0;
}
```

הגדרות שקולות

```
void init_array(int *array, int size)
{
    int *ptr;
    for (ptr = array; ptr < &array[size]; ptr++)
        *ptr = 0;
}
```

מצביעים ומחרוזות

- מימוש strlen בעזרת מצביע

```
int strlen(const char* str)
{
    int len = 0;

    if (str == NULL)
        return -1; /* Error! */

    while (*str != '\0') {
        len++;
        str++;
    }

    return len;
}
```

חיפוש תו במחרוזת

- הפונקציה strchr מחפשת תו במחרוזת
- מוגדרת ב string.h
- מחזירה מצביע למופע הראשון של התו או NULL אם לא קיים

```
char* strchr(const char* str, char c)
{
    if (str == NULL)
        return NULL;

    while (*str != '\0') {
        if (*str == c)
            return str;
        str++;
    }

    return NULL;
}
```

הפונקציה strstr

- חיפוש תת-מחרוזת

```
char* strstr(const char *haystack, const char *needle)
{
    int needlelen;

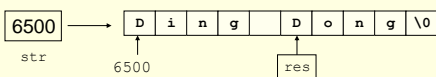
    if (*needle == '\0')
        return (char *) haystack;

    needlelen = strlen(needle);
    for (; (haystack = strchr(haystack, *needle)) != NULL; haystack++)
        if (strncmp(haystack, needle, needlelen) == 0)
            return (char *) haystack;

    return NULL;
}
```

הפונקציה strstr

```
char str[] = "Ding Dong";
char *res = strstr(str, "Don");
```



```
printf("res=%p\n", res);    res=6505
printf("res=%s\n", res);   res=Don
```

דוגמא – כמה פעמים מופיעה מילה בשי

```
int main()
{
    const char *rhyme = "Humpty Dumpty sat on a wall,\n "
                       "Humpty Dumpty had a great fall.\n "
                       "All the king's horses,\n "
                       "And all the king's men,\n "
                       "Couldn't put Humpty together again.\n";

    const char *humpty = "Humpty";
    char *ptr = NULL;
    int count = 0;

    for (ptr = strstr(rhyme, humpty); ptr != NULL;
         ptr = strstr(ptr + 1, humpty)) {
        count++;
    }
    printf("The string %s appears %d times\n", humpty, count);
    return 0;
}
```



תרגיל לבית: strcpy בשורה אחת

```
char* strcpy(char *dst, const char *src)
{
    char *s = dst;
    while (*dst++ = *src++);
    return s;
}
```