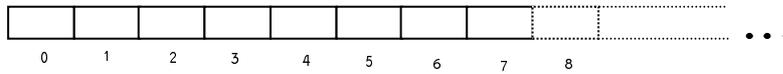


תכנות ב C - מצביעים וניהול זיכרון

זיכרון המחשב

ניתן לחשוב על זיכרון המחשב בצורה מופשטת כסדרה של תאים הנקראים בתים (bytes):

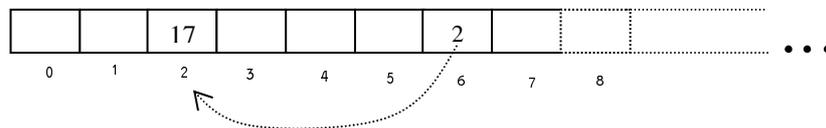


כל byte בנוי משמונה יחידות בינאריות הקרויות סיביות (bits). נהוג לסמן את שני המצבים האפשריים של סיבית ב- 0 ו- 1. מבחינה פיזית, שני מצבים שונים אלה יכולים להיות ממומשים בדרכים שונות. מספר המצבים השונים בהם יכל byte להימצא הוא $2^8 = 256$. כל byte נימצא במקום מוגדר בסדרה הניתן לאפיון ע"י מספר סידרתי. למספר המציין את מיקום ה- byte בזיכרון ניקרא הכתובת של ה- byte (באיור, המספרים הקטנים המופיעים מתחת לתאים הם הכתובות שלהם). כל אינפורמציה שאנו שומרים בזיכרון **מקודדת** בייצוג בינארי ונשמרת בתא אחד או יותר. ב C, האופרטור sizeof, מאפשר לדעת את מספר ה- bytes הדרושים לאחסון משתנה מסוים. לאחסון משתנה מטיפוס char, דרוש תמיד byte בודד. לאחסון משתנה מטיפוס int ידרשו בד"כ מספר תאים (על הפלטפורמה שלנו, ארבעה).

לכל משתנה (או ביטוי) ב- C יש **ערך וטיפוס**. הערך הוא מצב הסיביות באזור המוקצה למשתנה והטיפוס הוא המשמעות או האינטרפרטציה של הערך. בדומה ליחידות פיזיקליות בהם לכל ערך יש משמעות (5 קילומטר, 5 דקות, 5 ניוטון וכו'), גם ב- C, אם byte מסוים מחזיק את הערך 101, יש טעם לשאול מה משמעות המספר או איזה מהות הוא מייצג. המשמעות של אותו ערך (או מצב סיביות) בזיכרון יכולה להיות שונה לחלוטין בהתאם לטיפוס.

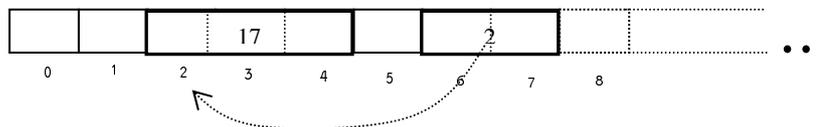
מצביעים

ערכים הנשמרים בזיכרון המחשב יכולים לייצג מהויות שונות. ראינו כבר שאפשר לשמור ערך המבטא מספר טבעי או ערך המבטא תו אלפביתי. בשפת C ניתן לשמור בזיכרון ערך המבטא **כתובת** בזיכרון. במילים אחרות, ניתן להגדיר משתנה המכיל מספר שמשמעותו, מיקום של משתנה אחר בזיכרון. למשתנה כזה קוראים **מצביע**. האיור הבא מדגים את המצב:



באיור זה, תא מספר 6 מכיל את מספר שמשמעותו "כתובתו של תא מס' 2". נוהגים להגיד שהמשתנה המאוכסן בתא 6, **מצביע** על המשתנה המאוכסן בתא 2 או שהמשתנה בתא 2 **מוצב** ע"י המשתנה בתא 6.

במציאות בד"כ מוקצה למצביע יותר מ- byte אחד והכתובת שהוא מכיל היא כתובת ה- byte הראשון של המשתנה עליו הוא מצביע:



הרעיון הפשוט של משתנה המכיל ערך שהוא כתובת של משתנה הוא רעיון יסודי מאד העומד בבסיס יצירת כל מבנה נתונים לא טריוויאלי. בזכות רעיון זה ניתן לייצג מבני נתונים בעלי אופי לא סידרתי בזיכרון סידרתי.

הצהרה על משתנה p כמצביע int תעשה באופן הבא:

```
int* p;
```

או:

```
int *p;
```

הטיפוס של p הוא int* כלומר, משמעות הערך ש p יקבל היא כתובת של משתנה מסוג int.

האופרטור & מאפשר לקבל את הכתובת של משתנה קיים. האופרטור * מאפשר להתייחס למשתנה המוצבע ע"י מצביע. התוכנית הבאה מדגימה שימוש פשוט במצביע:

```
int main() {
    int *p;
    int n = 7;
    p = &n;
    printf("the address of n is %d and the value of n is %d\n", (int)p, *p);
    *p = *p+*p;
    printf("the address of n is %d and the value of n is %d\n", (int)p, *p);

    printf("the conventional way to print an address is %p", p);
    return 0;
}
```

הסבר:

בשורה הראשונה הגדרנו משתנה בשם p מטיפוס int* - מצביע ל-int. בשורה השנייה הגדרנו את המשתנה n מטיפוס int ואתחלנו אותו ל-7. ניתן היה לכתוב את שתי ההצהרות בשורה אחת, באופן הבא:

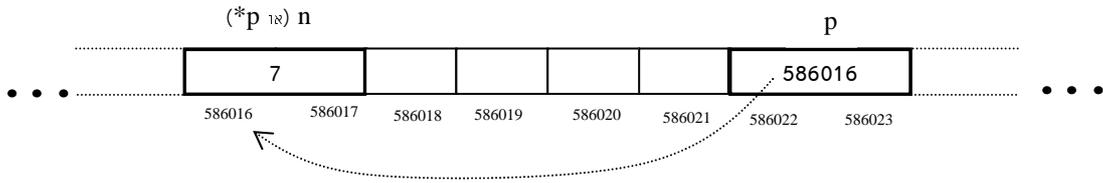
```
int *p, n=7;
```

הכתיב הבא הוא שקול אך מבלבל:

```
int* p, n=7;
```

(הטיפוס של n הוא עדיין int ולא int*)

בשורה השלישית, שערכנו את הביטוי &n. ערך הביטוי הוא הכתובת של המשתנה n בזיכרון וטיפוס הביטוי (משמעות הערך) הוא int* (כתובת של int). את הערך הזה השמנו, במשתנה p. יש לשים לב שהשמנו למשתנה מטיפוס int* ערך מטיפוס int. לאחר ביצוע פקודה זו, מצב הזיכרון יהיה כזה:



(בשרטוט זה קיימת הנחה ש: sizeof(int*)=sizeof(int)=2)

השורה הרביעית היא שורת הדפסה. הערך הראשון המודפס הוא הערך של p מומר לערך מטיפוס int. המרת טיפוסים נקראת casting ונעשית לפי התחביר הבא:

```
(type)expression
```

במקרה שלנו המרנו מספר המייצג כתובת של int למספר int, אותו ניתן להדפיס באופן רגיל. הערך השני שהודפס הוא הערך של המשתנה המוצבע ע"י p - n. בהנחה ש n באמת ממוקם כמו שמראה האזור, הפלט כתוצאה משורה זו, יהיה:

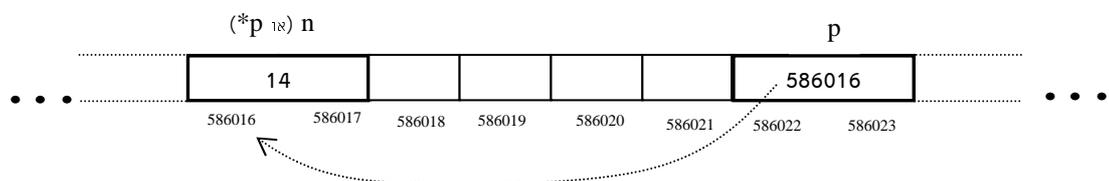
```
the address of n is 586016 and the value of n is 7
```

בשורה החמישית אנחנו עובדים עם המשתנה המוצבע ע"י p. הביטוי *p+*p הוא חיבור של ערך המשתנה המוצבע ע"י p, לעצמו. ערך זה מושם למשתנה המוצבע ע"י p. הקוד:

```
n = n+n;
```

הוא קוד שקול לשורה זו.

כעת, הזיכרון נראה כך:



הפלט של השורה השישית, יהיה אם כן :

the address of n is 586016 and the value of n is 14

בשורה השביעית אנו מדפיסים את הערך של p בצורה המקובלת. הסימון %p במחרוזת של printf קובע שיש להדפיס כתובת. כתובת מודפסת בייצוג הקסדצימאלי מסיביות פרקטיות :
the conventional way to print an address is 8f120

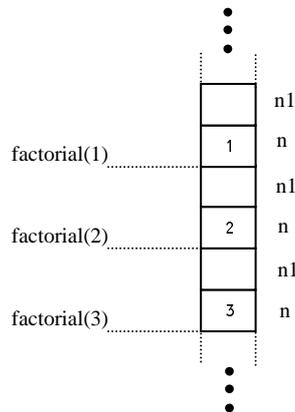
הקצאות על המחשנית והקצאות על הערמה

בדוגמאות שראינו עד כה, כל המשתנים בהם השתמשנו היו מקומיים, לוקאליים. משך הקיום של משתנים מקומיים הוא הזמן שהפונקציה שבה הם נמצאים, פועלת. נתבונן בקוד הבא :

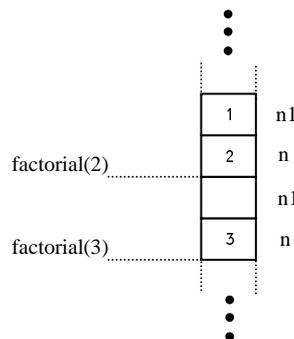
```
int factorial(int n) {
    int n1;
    if(n<=1)
        return 1;
    n1 = factorial(n-1);
    return n1*n;
}

int main() {
    printf("factorial(3) = %d", factorial(3));
    /* n and n1 are not known here */
    return 0;
}
```

המשתנים n ו n1 הם משתנים לוקאליים. משתנים לוקאליים מוקצים על אזור מיוחד בזיכרון המנוהל בצורה של מחשנית. בתוכנית הספציפית ייווצרו בשלב מסוים 3 עותקים של משתנים אלה על המחשנית :



הסיבה שציירנו את תאי הזיכרון במאונך היא רק לשם ההמחשה שמדובר במחשנית. למעשה מדובר בקטע מאותו זיכרון סידרתי עליו דיברנו עד כה. לאחר שהפונקציה factorial(1) תסתיים, ו ל - n1 של factorial(2) יושם ערך, מצב הזיכרון יהיה כזה :



$n!$ - n , המשתנים הלוקאליים של $\text{factorial}(1)$ כבר אינם קיימים.

במקרים רבים יש צורך בהקצאת מקום בזיכרון שמשך הקיום שלו לא מוגבל לזמן פעולת הפונקציה. כל מיקרה בו פונקציה אמורה לתחזק מבנה נתונים של התוכנית כולה, הוא מקרה כזה. הקצאה כזו נקראת **הקצאה דינמית** והצורה המקובלת לבצע אותה ב-C, היא בעזרת הפונקציה `malloc`. הקוד הבא מדגים זאת:

```
#include <stdio.h>
#include <stdlib.h>

int* foo() {
    int *p;
    p = (int*)malloc(sizeof(int));
    if(!p) {
        printf("allocation problem\n");
        exit(1);
    }
    *p = 17;
    return p;
}

int main() {
    int* p1 = foo();
    printf("p1 = %d\n", *p1);
    free(p1);
    return 0;
}
```

הסבר:

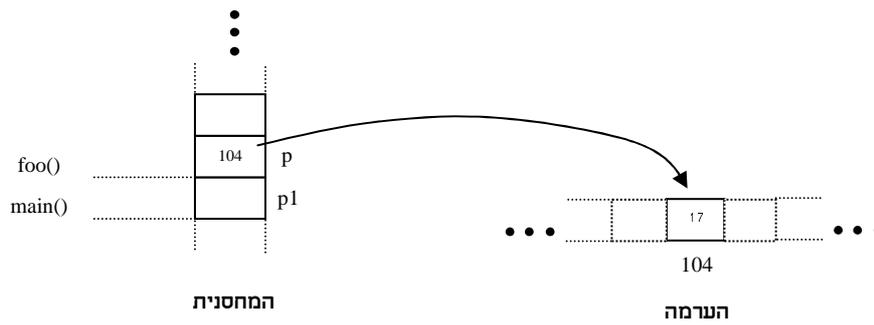
בכדי להשתמש בפונקציות `malloc` ו-`free`, יש לכלול את הקובץ `***malloc.h` ו-`***stdlib.h`. הקובץ כולל שירותים רבים נוספים בהם לא נשתמש כרגע. הפונקציה `foo` מצהירה על מצביע לוקאלי בשם `p`. ל-`p` מושם הערך החוזר מהפונקציה `malloc`. הפונקציה `malloc` מקצה זיכרון בצורה דינמית ומחזירה את כתובת הזיכרון בו בוצעה ההקצאה. `malloc` מקבלת כפרמטר את מספר ה-byte-ים שעליה להקצות, היא מקצה זיכרון רציף בגודל המבוקש באזור זיכרון המכונה "ערמה" ומחזירה את הכתובת של ה-`byte` הראשון שהקצתה. מכיוון ש-`malloc` איננה יודעת לאיזה טיפוס ישמש הזיכרון שהקצתה, היא לא יכולה לדעת מה צריך להיות הטיפוס המדויק של הכתובת שהיא מחזירה (האם זה למשל `int*` או `double*`). בגלל סיבה זו, `malloc` מחזירה כתובת כללית, בלי להתחייב כתובת של מה היא. הטיפוס שהיא מחזירה הוא `void*` שמשמעו, כתובת לטיפוס לא ידוע. בשביל להשתמש בכתובת מטיפוס `void*` יש לבצע המרה (casting) שתגדיר מהו הטיפוס המוצב. במקרה שלנו הטיפוס המוצב הוא `int`, לכן בצענו המרה מ-`void*` ל-`int*`. הפרמטר ששלחנו ל-`malloc` היה מספר הבתים הדרושים לאחסון `int` בודד.

אזור הזיכרון המכונה ערמה, אינו מנוהל כמו המחסנית. זיכרון שהוקצה על הערמה נישאר עד שהוא משוחרר ע"י הפונקציה `free`.

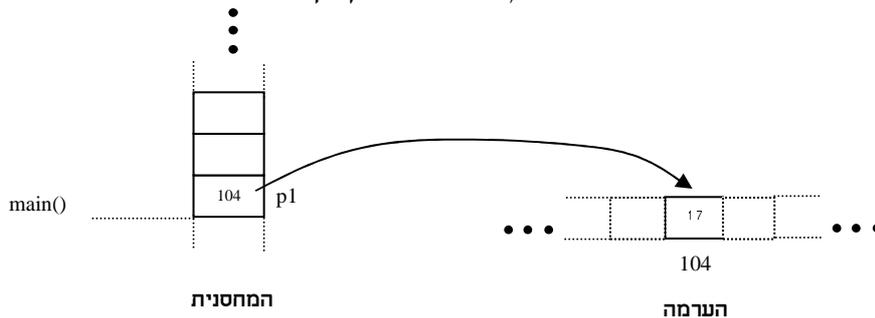
הקצאה דינמית עלולה להיכשל. יכל להיות שבזמן בקשת ההקצאה לא היו למערכת משאבי הזיכרון הדרושים או שיש בעיית חומרה. בכל מיקרה של כישלון בהקצאה, יחזיר `malloc` את הערך 0 (NULL). הפקודה הבאה בודקת אם ההקצאה נכשלה. במקרה של כישלון, מודפסת הודעת שגיאה ומבוצעת יציאה מבוקרת מהתוכנית. הפונקציה `exit`, נמצאת ב-`stdlib` וגורמת ליציאה מהתוכנית והחזרת ערך שגיאה למערכת ההפעלה. במקרה הזה החזרנו את הערך 1 (השונה מ-0, תקין).

בהנחה שההקצאה הדינמית הצליחה, התוכנית משימה באזור שהוקצע את הערך 17 ולאחר מכן מחזירה את הכתובת של הזיכרון הדינמי כערך מוחזר מהפונקציה `foo`. הפונקציה `main` מפעילה את הפונקציה `foo` ושומרת את הערך המוחזר ממנה במצביע `p1`. הפקודה הבאה של `main` מדפיסה את הערך של המשתנה המוצב ע"י `p`. ערך זה הוא עדיין 17 למרות שהפונקציה שבה הושם ערך זה כבר לא פעילה. לפני סיום התוכנית משוחרר הזיכרון שהוקצע דינמית. הפונקציה `free` מקבלת מצביע לאזור שהוקצה דינמית ודואגת לשחררו. תוכנית שאינה משחררת את כל הזיכרון שהקצתה נחשבת לא תקינה ועלולה לגרום לבעיות ברמה של הגורם שהריץ אותה.

האיור הבא ממחיש את מצב הזיכרון ממש לפני שהפונקציה `foo` מסתיימת:



כאשר הבקרה מגיעה לשורה השניה של main, יראה הזיכרון כך :



את הערמה בחרנו לצייר בצורה אנכית בשביל להבדיל אותה מהמחסנית. סדר ההקצאה על הערמה אינו תלוי בנו ואיננו יכולים להניח הנחות לגביו. במציאות, כזכור, מהווים גם הערמה וגם המחסנית אזורים שונים של אותו זיכרון סידרתי יחיד.

כפי שראינו, האחריות על שחרור זיכרון שהוקצה דינמית היא על המתכנת. עובדה זו היא עד הבדל משמעותי בין C/C++ ל Java. ב Java קיים מנגנון אוטומטי לשחרור זיכרון, מנגנון הנקרא garbage collection. המנגנון מזהה אזור בזיכרון שכבר איננו מוצבע, ומשחרר אותו באופן אוטומטי. למרות הקיום מנגנון כזה מקל על המתכנת, יש לו עלות מבחינת ביצועים. המנגנון דורש בדיקות תקופתיות במהלך התוכנית וגם זיכרון נוסף עבור נתוני המנגנון. במקרים רבים המנגנון איננו מושלם.

אי שחרור זיכרון יכול לגרום למחסור בזיכרון בזמן ריצת התוכנית ולבעיות לאחר סיומה.

מערכים ומצביעים

ב C מערך הוא פשוט רצף של תאים בזיכרון. שם המערך הוא מצביע לתא הראשון שלו. הקוד הבא מדגים זאת :

```
int main() {
    float arr[20];
    arr[0] = 2;
    printf("The value of the first cell in the array is: %f\n",*arr);
    arr[7] = 17;
    printf("The value of the 8th cell in the array is: %f\n",*(arr+7));
    return 0;
}
```

הערכים בפלט יהיו 2 ו-17.

הקוד הבא מבצע פעולה דומה רק עם שימוש בהקצאה דינמית :

```
int main() {
    float *arr = (float*)malloc(20*sizeof(float));
    arr[0] = 2;
    printf("The value of the first cell in the array is: %f\n",*arr);
    arr[7] = 17;
    printf("The value of the 8th cell in the array is: %f\n",*(arr+7));
}
```

```

free(arr);
return 0;
}
יש לשים לב שכאשר חיברנו לכתובת שב - arr 7, לא הוספנו לכתובת את המספר 7 אלא הוספנו
את המספר הדרוש להתקדמות ב 7 תאים מסוג float. הערך המספרי שהוסף, אם כן, הוא
: 7*sizeof(float). הקוד הבא מראה זאת:

```

```

int main() {
float arr[20];
printf("The actual difference in bytes between arr[7] and arr[0] is %d = %d * %lu \n", (int)(arr+7)-
(int)arr, 7, sizeof(float));
printf("The same difference in float units is %ld\n", (arr+7)-arr);
return 0;
}

```

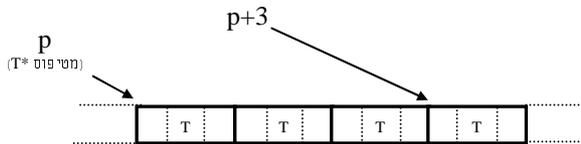
פלט:

The actual difference in bytes between arr[7] and arr[0] is 28 = 7 * 4
The same difference in float units is 7

את ההתנהגות הזאת ניתן לנסח כחוק כללי לגבי אריתמטיקה של מצביעים:

כל פעולה אריתמטית בין מצביע לטיפוס T, לבין מספר שלם, מחושבת ביחידות של sizeof(T).

איור:



הסיבה להתנהגות זו היא השימוש הנפוץ במצביעים בכדי לבצע מניפולציות במערכים.

נראה עד דוגמה לכלל זה:

```

int main() {
double *ptr=(double*)17;
ptr++;
printf("%d\n", (int)ptr);
ptr-=2;
printf("%d\n", (int)ptr);
return 0;
}

```

במערכת בה sizeof(double)=8, נקבל את הפלט:

25
9

מצביעים למצביעים

כאמור, מצביע הוא משתנה המכיל כתובת של משתנה. אותו משתנה מוצבע, יכל עקרונית גם הוא להיות מצביע. לפי הדוגמאות שראינו עד כה, מצביע למשתנה מטיפוס T, טיפוסו יהיה T*. לפי אותו הגיון, מצביע המצביע למשתנה מסוג T*, טיפוסו יהיה T**. עד עכשיו ראינו שהאופרטור * הפועל על מצביע p, מחזיר את המשתנה עליו p מצביע. אם אותו משתנה גם הוא מצביע, אז הביטוי **p יחזיר את המשתנה עליו מצביע *p. נתבונן בקוד הבא:

```

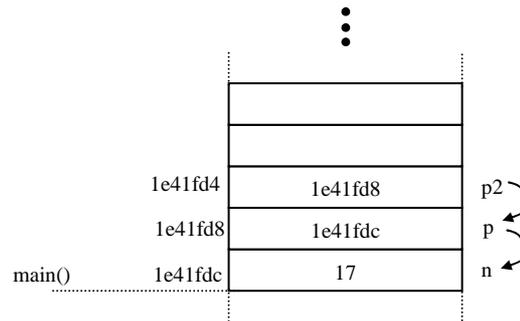
int main() {
int n = 17;
int *p = &n;
int **p2 = &p;
printf("the address of p2 is %p \n", &p2);
printf("the address of p is %p \n", p2);
printf("the address of n is %p \n", *p2);
printf("the value of n is %d \n", **p2);
return 0;
}

```

הפלט יכל להראות כך :

the address of p2 is 1e41fd4
the address of p is 1e41fd8
the address of n is 1e41fdc
the value of n is 17

לאחר ההשמות, יראה זיכרון המחשב כך :



(המספרים בדוגמה זו מראים שבפלטפורמה עליה הורצה התוכנית, המחשנית גדלה לכיוון תחלית הזיכרון)

חשוב להבין שגם מצביע למצביע, בדיוק כמו מצביע רגיל, מכיל פשוט כתובת - מספר המבטא מיקום בזיכרון. ההבדל היחידי בין מצביע רגיל למצביע למצביע, הוא הטיפוס כלומר המשמעות של הערך המוכל בו. כאשר מדובר במצביע למצביע משמעות הערך היא כתובת של מצביע. כאשר מדובר במצביע רגיל, משמעות הערך היא כתובת של משתנה רגיל. בגלל העובדה שבשני המקרים מכילים המשתנים כתובות, המקום הדרוש לאחסונם הוא זהה. במילים אחרות, על כל פלטפורמה יתקיים:

`sizeof(int*)=sizeof(int**) = sizeof(double*) = sizeof(double****) = sizeof(void*)`

כצפוי, ניתן להכליל את כל מה שנאמר לגבי מצביע למצביע, ולהשתמש במצביע למצביע למצביע וכן הלאה.

הקוד הבא מדגים שימוש במצביעים למצביעים בהקשר של מערכים :

```
#define LENGTH 5

int** makeArray() {
    int **arr2d = (int**)malloc(sizeof(int*)*LENGTH),i,j;
    for(i=0; i<LENGTH; i++) {
        arr2d[i] = (int*)malloc(sizeof(int)*(i+1));
        for(j=0; j<=i; j++)
            arr2d[i][j] = i*j; /* same as: *((arr2d+i)+j) = i*j */
    }
    return arr2d;
}

int main() {
    int **ar = makeArray(),i,j;
    for(i=0; i<LENGTH; i++) {
        for(j=0; j<=i; j++)
            printf("%d ",*((ar+i)+j)); /* same as: printf("%d ",ar[i][j]); */
        printf("\n");
    }
}
```

```

return 0;
}

```

פלט :

```

0
0 1
0 2 4
0 3 6 9
0 4 8 12 16

```

מצביעים כפרמטרים לפונקציות

התבוננו בקוד הבא :

```

void foo(int n) {
    n++;
    printf("n=%d\n",n);
}

```

```

int main() {
    int n=7;
    foo(n);
    printf("n=%d\n",n);
    return 0;
}

```

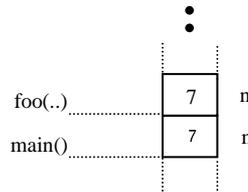
הפלט שלו הוא :

```

n=8
n=7

```

הסיבה לכך היא שהמשתנה n של foo הוא העתק של המשתנה m של main :



כאשר פרמטר המתקבל הוא העתק של הפרמטר שנישלח, העברת הפרמטר נקראת העברה by value.

כל שינוי שנעשה בהעתק של הפרמטר, אינו משפיע כמובן על המשתנה המקורי. נניח שאיננו רוצים העברה by value אלא רוצים שכל השינויים המתבצעים בתוך הפונקציה, ישפיעו על המשתנה המקורי. איך ניתן לעשות זאת ?

ב-C, ניתן לעשות זאת בעזרת מצביעים. נשכתב את התוכנית כך :

```

void foo(int* p) {
    (*p)++;
    printf("n=%d\n",*p);
}

```

```

int main() {
    int n=7;
    foo(&n);
    printf("n=%d\n",n);
    return 0;
}

```

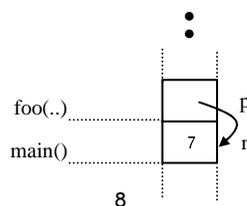
כעת, הפלט יהיה :

```

8
8

```

מבחינת הזיכרון, התמונה תהיה כזו :



ניתן לראות כי n ו- p הם שמות נרדפים לאותו משתנה.

דוגמה נפוצה מאד להעברה של פרמטרים בצורה כזו היא הפונקציה `swap`, המקבלת שני משתנים ומחליפה את ערכיהם:

```
void swap(int* p1, int* p2) {
    int tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}
```

```
int main() {
    int a=2,b=5;
    printf("%d %d\n",a,b);
    swap(&a,&b);
    printf("%d %d\n",a,b);
    return 0;
}
```

פלט:

```
2 5
5 2
```

מובן שמימוש נאיבי כמו המימוש הבא לא היה משיג את המטרה:

```
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

בכל מיקרה שבו פונקציה אמורה להחזיר יותר מערך אחד, אפשר להשתמש באותה שיטה. נתבונן לדוגמה על פונקציה המחשבת שני שורשים של משוואה ריבועית:

```
void roots(double a, double b, double c, double* pz1, double* pz2) {
    double discriminant = sqrt(b*b-4*a*c);
    *pz1 = (-b+discriminant)/(2*a);
    *pz2 = (-b-discriminant)/(2*a);
}
```

```
int main() {
    double z1,z2;
    roots(2,-6,4,&z1,&z2);
    printf("roots: %f %f\n",z1,z2);
    return 0;
}
```

פלט:

```
roots: 2.000000 1.000000
```

כדאי לשים לב שגם כאשר מעבירים מצביע כפרמטר, המצביע עצמו עובר `by value`, לדוגמה:

```
void foo(int* p) {
    p++;
    printf("%d\n",(int)p);
}
```

```
int main() {
    int *p = (int*)17;
    foo(p);
    printf("%d\n",(int)p);
}
```

```
return 0;
}
```

פלט :

```
21
17
```

מצביע ל - NULL

הדרך המקובלת לסמן אי-הצבעה של מצביע היא לתת לו להצביע על הכתובת 0. המערכת מבטיחה ש - 0 איננה כתובת לגיטימית. הקוד באה מדגים שימוש פשוט :

```
void foo(int* p) {
    if(p!=0) /* equal to "if(p)" */
        printf("p points to something\n");
    if(p==0) /* equal to "if(!p)" */
        printf("p doesn't point to anything\n");
}
```

```
int main() {
    int *p =0,i;
    foo(p);
    p = &i;
    foo(p);
}
```

פלט :

```
p doesn't point to anything
p points to something
```

כזכור, כל ערך השונה מ - אפס נחשב גם כביטוי בולאני בעל ערך true והערך אפס נחשב כ - false.

בקובץ stdio.h קיימת הגדרה של NULL להיות 0. נהוג להשתמש ב - NULL במקום כתיבה מפורשת של הערך 0 כאשר עובדים עם מצביעים. בדוגמה האחרונה, ניתן להחליף את כל המופעים של 0 ב - NULL.

מחרוזות

מחרוזות ב - C מיוצגות ע"י מערך של char שבסופו הערך 0. הקוד הבא מדגים זאת :

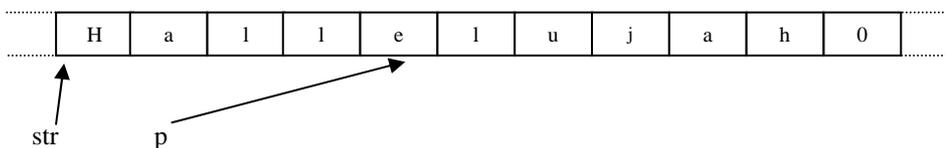
```
#include <stdio.h>
```

```
int main() {
    int i=0;
    char *str = "Hallelujah",*p=str;
    while(*p) {
        printf("%c",*p);
        p++;
    }
    printf("\n");
    while(str[i])
        printf("%c",str[i++]);
    printf("\n%s\n",str);
    return 0;
}
```

פלט :

```
Hallelujah
Hallelujah
Hallelujah
```

תמונת הזיכרון של התוכנית היא זו :



האותיות כמובן מקודדות כמספרים. הערך 0 הוא byte שכל הסיביות שלו הן 0.

בדוגמה שראינו, אזור הזיכרון בו אוכסנה המחרוזת איננו המחסנית ואיננו הערמה, אלא מקום מיוחד המיועד למחרוזות הרשומות בתוך התוכנית. מקום זה אין צורך להקצות או לשחרר והוא נשאר תקף לאורך כל ריצת התוכנית. אם היינו רוצים שהמחרוזת תהיה על המחסנית היינו יכולים להחליף את השורה השניה של - main בשורה הבאה :

```
char str[] = {'H','a','l','l','e','l','u','j','a','h','\0'}, *p=str;
```

הביטוי 'H' ערכו הקידוד של התו H וטיפוסו char. הביטוי '\0' ערכו 0 וטיפוסו char.

אם היינו רוצים שמחרוזת תישמר על הערמה, יכולנו להחליף שורה זו בשורות :

```
char *s = "Hallelujah", *str = (char*)malloc(strlen(s)+1), *p=str;
strcpy(str,s);
```

בכדי שאפשר יהיה להפעיל את הפונקציות strlen ו-strcpy יש להכליל את הקובץ string.h ע"י הפקודה הבאה בתחילת התוכנית :

```
#include <string.h>
```

הפונקציה strlen מחזירה אורך של רשימה נתונה. אורך של רשימה הוא מספר התווים המרכיבים אותה, בלי לספור את התו '\0' המסמן את סופה. הפונקציה strcpy מאפשרת העתקת מחרוזות. קיימת פונקציה שימושית נוספת המאפשרת השוואה לקסיקוגרפית בין שתי מחרוזות ששמה strcmp. לפרטים נוספים פנו ל - man.

אם בחרנו שהמחרוזת תמוקם על זיכרון שהקצנו על הערמה, יש לשחרר אותו בסיום התוכנית : free(str);

struct

ב - C, ניתן להצהיר על טיפוס חדש המהווה קיבוץ של כמה טיפוסים מוכרים. טיפוס כזה נקרה struct (עבור structure - מבנה). משתנה מטיפוס ה - struct יכול בתוכו משתנים מטיפוסים מוכרים. הדבר דומה למושג ה-class המוכר מ - java אך אינו כולל פונקציות (מתודות). לדוגמה :

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
struct Person {
    unsigned int _id;
    char *_name;
    float _age;
}; // note that the semicolon is mandatory
```

```
void printPerson(Person per) {
    printf("%d, %s, %f\n", per._id, per._name, per._age);
}
```

```
int main() {
    Person per1, *perPtr;
    per1._id = 14157689; per1._name = "Hizkiya";
    per1._age = 17.5;
    perPtr = (Person*)malloc(sizeof(Person));
    (*perPtr)._id = 177777;
    perPtr->_name = "Yirmiya"; /* same as (*perPtr)._name = "Yirmiya"; */
    perPtr->_age = .5;
    printPerson(per1);
    printPerson(*perPtr);
}
```

```

    free(perPtr);
    return 0;
}

```

פלט :

```

14157689, Hizkiya, 17.500000
177777, Yirmiya, 0.500000

```

שימו לב שכאשר כתבנו

```
(*perPtr)._id = 177777;
```

השימוש בסוגריים היה הכרחי. אילו היינו משמיטים אותן, הקומפיילר היה מבין את השורה כך :

```
*(perPtr._id) = 177777;
```

פרוש כזה גורר כמובן שגיאת קומפילציה.

מכיוון שפניה למשתנה של מבנה מוצבע היא פעולה נפוצה מאד ומכיוון שהשימוש בסוגריים מסרבל את הכתיבה, סיפקו מתכנני השפה דרך מקוצרת :

```
perPtr->_id = 177777;
```

שימוש בצורה המקוצרת לא משנה את המשמעות.

רשימה משורשרת ב - C

קעת יש בידינו מספיק ידע ב - C בכדי לכתוב קוד למבני נתונים שונים. נדגים קוד של רשימה משורשרת :

```

#include <stdio.h>
#include <stdlib.h>

```

```

struct Node {
    double _data;
    Node *_next;
};

```

```

struct List {
    Node *_head;
    unsigned int _size;
};

```

```

Node* createNode(double data, Node* next) {
    Node *p = (Node*)malloc(sizeof(Node));
    p->_data = data;
    p->_next = next;
    return p;
}

```

```

List* createList() {
    List *p = (List*)malloc(sizeof(List));
    p->_head = NULL;
    p->_size = 0;
    return p;
}

```

```

void insertFirst(List* p, double data) {
    p->_head = createNode(data,p->_head);
    p->_size++;
}

```

```

void printList(List* listPtr) {
    Node *p=listPtr->_head;
    while(p) {
        printf("(%f)->",p->_data);
        p = p->_next;
    }
}

```

```

    printf("|| size:%u \n",listPtr->_size);
}

List* cloneList(List* listPtr) {
    List *ret = createList();
    Node *old = listPtr->_head, **copy = &(ret->_head);
    ret->_size = listPtr->_size;
    while(old) {
        *copy = createNode(old->_data,NULL);
        old = old->_next;
        copy = &((*copy)->_next);
    }
    return ret;
}

void freeList(List *listPtr) {
    Node *p1 = listPtr->_head,*p2;
    while(p1) {
        p2 = p1;
        p1 = p1->_next;
        free(p2);
    }
    free(listPtr);
}

int main() {
    int array[]={2,4,6,12,77,99},i; // we used a short way to initialize a local array
    List *listPtr1 = createList(),*listPtr2;
    for(i=0; i<6; i++)
        insertFirst(listPtr1,array[i]);
    printList(listPtr1);
    listPtr2 = cloneList(listPtr1);
    listPtr1->_head->_next->_data = 1.1111;
    printList(listPtr1);
    printList(listPtr2);
    freeList(listPtr1);
    freeList(listPtr2);
    return 0;
}

```

פלט :

```

(99.000000)->(77.000000)->(12.000000)->(6.000000)->(4.000000)->(2.000000)->|| size:6
(99.000000)->(1.111100)->(12.000000)->(6.000000)->(4.000000)->(2.000000)->|| size:6
(99.000000)->(77.000000)->(12.000000)->(6.000000)->(4.000000)->(2.000000)->|| size:6

```

מספר הערות נוספות

מצביעים ב Java

לאחר שמושג המצביע הוסבר, ניתן להסביר מספר תופעות ב - Java שאולי לא היו מובנות לחלוטין עד כה. נתבונן בקוד ה - Java הבא :

```

public class Main {
    static private void foo1(int n) {
        n = 8;
    }
    static private void foo2(A a) {
        a._data = 8;
    }
    public static void main(String[] args) {
        A a = new A();
        int n;
    }
}

```

```

    a._data = n = 7;
    foo1(n);
    foo2(a);
    System.out.println("a._data="+a._data+", n="+n);
}
}

class A {
    public int _data;
}

```

הפלט יהיה :

a._data=8, n=7

מדוע a._data שונה ע"י foo2 ו-n לא שונה ע"י foo1 ?
 התשובה היא שב Java, כל שם של אובייקט ממומש ע"י מצביע לאובייקט. במקרה זה a הוא מצביע לאובייקט מסוג A. כאשר a הועבר ל foo2, הועבר המצביע by value כך ש foo2 החזיקה העתק של המצביע לאובייקט A המקורי. השינוי ש foo2 עשתה, היה באובייקט עצמו ולא במצביע כך שהאובייקט המקורי עבר שינוי. לעומת זאת המשתנה n מטיפוס int הועבר by value ל foo1, כך ש foo1 החזיקה העתק. כל שינוי של ההעתק לא השפיע על המשתנה המקורי.

אם נשנה את foo2 באופן הבא :

```

static private void foo2(A a) {
    a = new A();
    a._data = 8;
}

```

הפלט יהיה :

a._data=7, n=7

שוב, מכיוון שהמצביע עצמו הועבר by value כל שינוי של מצביע ההעתק לא ישנה את המצביע המקורי.

זירות במצביעים !

מצביעים הם כלי רב עוצמה המאפשר כתיבת קוד אלגנטי ויעיל. מצד שני, בעבודה עם מצביעים קל מאוד לטעות וליצור באגים קשים מאד לאיתור. בערך שני שליש מדמעות הסטודנטים הם תוצאה ישירה של טעויות מצביעים. שורש הבעיה הוא העובדה שתוכנית שגויה עלולה לעבוד בצורה תקינה. לדוגמה :

```
#include <stdio.h>
```

```

int main() {
    int *p; /* forgot to allocate space (use malloc) */
    p[5] = 12345;
    printf("%d\n",p[5]);
    return 0;
}

```

התוכנית הזאת יכולה להדפיס :

12345

או "לעוף" (להסתיים בצורה לא תקינה, "להתרסק", ליפול, "crash") בליווי הודעה כמו :

```
segmentation fault
```

השגיאה בתוכנית היא שלא בוצעה הקצעת מקום למערך בו אנו משתמשים. במה תלויה התנהגות התוכנית ? מדוע היא עלולה לעבוד ולהדפיס את הפלט הרצוי ?

הערך (הכתובת) שמכיל המשתנה p אינו מוגדר בתוכנית. בד"כ אנו מתייחסים לערך כזה כערך "זבל". אם במקרה אותו ערך "זבל" הוא כתובת אזור זיכרון בו ניתן לכתוב, התוכנית תעבוד, אם לא, היא תעוף. גם עם בהרצה מסוימת התוכנית תעבוד, בהרצה נוספת היא עלולה לקרוס. אין שום הנחה שאנו יכולים להניח על הערך (הלא מאותחל) של p, הוא יכל להשתנות בין הרצות ובין פלטפורמות. התוכנית הזאת היא בכל מיקרה שגויה, אך השגיאה לא תמיד תתגלה.

העובדה שתוכנית שגויה עלולה לתפקד כתקינה, היא בעיה רצינית. הדוגמה האחרונה נראית טריוויאלית אך נסו לדמיין שזו הייתה פונקציה המהווה חלק מקוד גדול המשתמש במערך.

התוכנית הגדולה הייתה עלולה להתרסק במקומות לגמרי לא צפויים. המערך שבו היא משתמשת נימצא במקום לא מוגדר וחלקים אחרים של התוכנית עלולים לכתוב עליו.

נביא עד שתי דוגמאות לשגיאות מצביעים:

```
#include <stdio.h>

int* foo() {
    int array[4] = {1,2,3,4};
    return array;
}

int main() {
    int *p = foo();
    printf("%d\n",p[3]);
    return 0;
}
```

התנהגות תוכנית איננה מוגדרת מכיוון ש foo מחזירה מצביע למערך לוקאלי לפונקציה. כאשר foo מסתיימת, הזיכרון בו נימצא array אינו חוקי לשימוש. שימוש ב -Wall - בקומפילציה יגרום להדפסת הזהרה לגבי הטעות הזאת.

דוגמה נוספת:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main() {
    char *p1 = "hi mom", *p2 = (char*)malloc(strlen(p1));
    strcpy(p2,p1);
    printf("%s\n",p2);
    return 0;
}
```

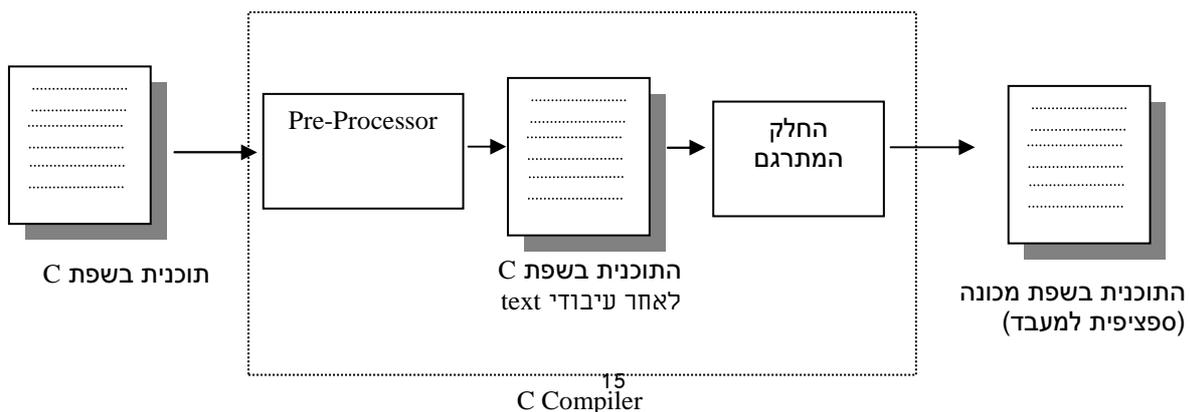
התנהגות תוכנית זו איננה מוגדרת מכיוון למערך המוצבע ע"י p2 לא הוקצע מספיק מקום בשביל להכיל את המחרוזת המוצבעת ע"י p2. יש לזכור שבכל מחרוזת קיים תו סיום וצריך לשמור מקום גם עבורו. פקודת ההקצאה הנכונה היא:

```
*p2 = (char*)malloc(strlen(p1)+1);
```

טעויות פשוטות וטיפשיות אלו, עלולות לגזול זמן תכנות ועצבים יקרים.

ה - pre-processor

קומפיילר הוא כידוע תוכנית מחשב המקבלת כקלט קובץ המכיל תוכנית בשפה אחת ומייצרת כפלט קובץ של תוכנית שקולה בשפה אחרת. במילים אחרות קומפיילר הוא תוכנה לתרגום שפת מחשב. קומפיילר של C (כמו gcc) מקבל קובץ בשפת C ומייצר קובץ בעל אותה משמעות בשפת מכונה. לפני שהקומפיילר מתחיל במלאכת התרגום עצמה, הוא מבצע עיבודי text פשוטים. החלק האחראי על עיבודים אלה ניקרא ה - pre-processor. ה - pre-processor מבצע פעולות של שתילת קבצים, החלפת מחרוזות, מחיקת קטעי קוד וכי. כאשר החלק המתרגם של הקומפיילר מתחיל לעבוד, הוא בעצם עובד על הפלט של ה - pre-processor. האיור הבא מדגים את יחסי הגומלין:



כל הפקודות המתחילות בתו '#' הן פקודות עבור ה - pre-processor. שתי דוגמאות לפקודות כאלה הן #define ו-#include. הפקודה #define old_string new_string גורמת להחלפה של כל מופע של old_string בקוד, ב new_string. הפקודה #include <file_name> גורמת לשתילת כל הקובץ ששמו file_name במקום הפקודה.

ניתן ליראות לראות את הפלט של ה - pre-processor ע"י הפעלה של gcc עם -E. לדוגמה, אם נכתוב קובץ כזה בשם check.c :

```
#include <stdio.h>
#define NUM 5
int main() {
    int x = NUM, y=NUM+1;
    return 0;
}
```

ואז נפעיל את הקומפיילר בצורה הבאה :

```
$ gcc -Wall -E check.c
```

נקבל את ההדפסה הבאה על המסך :

```
... כל הקוד שכתוב בקובץ stdio.h
int main() {
    int x = 5, y = 5+1;
    return 0;
}
```

כפי שניתן לראות, הפלט של ה - pre-processor הוא עדיין קוד תקני ב - C.

ב - C נהוג להשתמש ב - define בכדי להגדיר קבועים. חשוב לזכור ש - define לא גורם להקצאת מקום עבור משתנה אלא רק גורם להחלפת מחרוזות.