בס"ד

מבוא

נגדיר מהי תוכנית מחשב: באופן בלתי פורמאלי זוהי סדרת פקודות לביצוע עבור מחשב. סדרת פקודות זו מתחילה ממקום קבוע מראש ומסתיימת באמצעות פקודות סיום שנכתבות על ידי המתכנת.
נגדיר מהי תוכנית בשפת c: תוכנית בשפת c היא תוכנית אשר אוסף הפקודות שבה שייכות לשפה מוסכמת מראש - שפת c. אוסף פקודות זה מוכר למתכנת ולתוכנה מסוימת הנקראת "מהדר לשפת c" או קומפיילר c, אשר אחראי לתרגום התוכנית משפת c לתוכנית המכילה פקודות המוכרות למחשב.

היסטוריה

שפת c פותחה בראשית שנות ה-70 על ידי חברת bell , השפה במקורה נועדה למערכת ההפעלה unix אך במהרה היא התפשטה ופותחו מספר רב של גרסאות. בשנת 1983 הקים ansi , מכון התקנים האמריקאי ועדה לקביעת תקן מוסכם לשפה כדי למנוע את ההבדלים השונים בין הגרסאות, תקן זה נקרא ansi c וכל המהדרים כיום בנויים על פי תקן זה.

מדוע c ?

- C זו שפה יחסית קטנה וכך קלה לתכנות.
- C דומה ל pascal אך בעלת כוח רב יותר.
- C נכתב בשפת Unix
- C ניידת כלומר קוד שנכתב על מכונה אחת יכול לעבור ללא קושי אל אחרת.
- C מאפשרת למתכנת גישה לתוך הקרביים של המחשב רמת ה bit
- C זו שפה מודולארית וכך היא קלה להבנה וחסכונית (ניתן לקרוא לפונקציות מס פעמים עם ארגומנטים שונים).
- C מהווה את הבסיס לשפת c++
- תכנות פרוצדורלי כלומר התוכנית מחולקת למספר קטעי קוד קטנים הנקראים פונקציות, כל פונקציה מטפלת בבעיה מסוימת.
- מכילה מגוון של פקודות לשליטה על זרימת התוכנית באופן עקרוני המחשב מבצע את הפקודות לפי סדר כתיבתן אולם ניתן לשלוט על זרימתה של התוכנית על ידי פקודות ולגרום לביצוע מותנה של חלקים בתוכנית או ביצוע קטעים מסוימים מספר רב של פעמים ובכך לחסוך את כתיבת הקוד מספר פעמים ואחסונו על ידי שימוש בלולאות.
- C מכילה יחסית מספר קטן של פקודות ולכן המהדר של השפה קטן ופשוט ותוצאות ההידור יעילות.

תוכנית ראשונה בC-

דוגמא 1:

```
#include <stdio.h>

void main()
{
   printf ("This line will be printed");
}
```

התבנית הכללית של פונקצית:main

```
void main()
{
   first command;
   second command;
   .....
   last command;
}
```

נסכם: כל תוכנית תתחיל על ידי הכללה (include) של ספריות נחוצות (על פי דרישת התוכנית) לאחריה נקרא לפונקציה הראשית (main) נפתח בסוגריים מסולסלות ובתוכם נכתוב את כל הפקודות ,לאחר מבן נסגור את הפונקציה על ידי סוגר מסולסל שמאלי.

הערה:כל פקודה חייבת להסתיים בנקודה פסיק (;)

פונקציות ספרייה

פונקציות ספרייה הינן "קופסאות שחורות" המקבלות פרמטרים, מבצעות פעולות המוגדרות מראש ומחזירות ערך. הפרמטרים המועברים לפונקציה נרשמים בתוך סוגריים מימין לשם הפונקציה, פעולת הפונקציה מתבצעת כאשר התוכנית "מגיעה" לפונקציה ואל הערך המוחזר ניתן להתייחס כאילו הוא מופיע במקום הפונקציה לאחר סיום פעולתה - לדוגמא ראינו בתוכנית הראשונה שימוש בפונקציה printf המצויה בספרייה stdio
נדון בצורה מורחבת על פונקציות בפרק העוסק בנושא.

הספרייה שבה השתמשנו בתוכנית הראשונה היא stdio כל פעולת קלט/פלט של נתונים לתוכנית מבצעים בעזרת פונקציות ספרייה המטפלות בקלט/פלט ובשביל להשתמש בפונקציות אלו יש צורך בהכללת הספרייה stdio לתוכנית.

ראינו בתוכנית הראשונה פקודת הדפסה : ()printf פקודה זו הינה למעשה סדרת פקודות למחשב שאינן נראות למתכנת הגורמות לכך שכל מה שמופיע בין הסוגריים ובין המרכאות הכפולות יודפס למסך.
ישנן תוספות לפקודת ההדפסה ,תוספות אלו יכתבו בין המרכאות.

תוספות עיקריות ל printf פקודת הדפסה :
- \n מעבר לשורה חדשה (new line)
- \t הזחה במספר תווים - תלוי בברירת המחדל tab
- \r מעבר לשורה חדשה - בדיוק כמו הקשה על Enter
- \" גרשיים
- \' גרש
- \\ לוכסן הפוך אחד (\)

דוגמא 1.2

```
#include <stdio.h>

void main()
{
   printf ("This line will be printed \n");
}
```

ההבדל בין תוכנית זו לקודמת הוא התו \n שמופיע במחרוזת להדפסה. תו זה אינו מודפס למסך אלא מסמל לפונקציה printf לרדת שורה במסך ולהמשיך

דוגמא 2:

```
#include <stdio.h>

void main()
{
   printf ("first sentence \n");
   printf ("second sentence \n");
   printf ("third sentence \n");
}
```

כל משפט יתחיל בשורה חדשה עקב השימוש ב \n

דוגמא 3:

```
#include <stdio.h>

void main()
{
   printf ("This line shows how C works \n\r");
   printf("we moved to a new line \n");
   printf("\t End program");
}
```

הוספנו בתוכנית זו את השימוש בתו \t שמשמעו כפי שהזכרנו קפיצה ב tab

הערות

אנו יכולים להוסיף הערות לתוכניתנו, הערות אלה ישמשו אותנו בלבד ואת כל מי שיקרא את התוכנית שכתבנו (מומלץ להשתמש בהערות על מנת לתעד את אופייה של פונקציה או איזה שהוא קטע קוד בכך התוכנית הופכת למובנת יותר) המהדר לא יתייחס להערות וידלג עליהן.

ישנם שני סוגי הערות:
- // הערה בעבור שורה אחת.
- /* */ מספר רב של שורות.

בס"ד

דוגמא 4:

```
#include <stdio.h>

void main()
{
   printf("This program "); //this text will not be shown
   printf("show who to write comments");//this text also will not be shown
}
```

הדגמנו את השימוש בשני סוגי הערות ,הערה ראשונה (//) תחומה הוא שורה אחת בלבד אך ההערה השניה (/* */) תחומה מספר שורות עד ה */

בס"ד

משתנים

משתנה הוא מקום מוגדר בזיכרון היכול להכיל ערך מטיפוס מסוים כאשר סוג הטיפוס מוגדר בהצהרה על המשתנה. לכל משתנה יש שם ייחודי וערך אחד מוגדר. משתנה מאחסן נתונים וערכו יכול להיקבע ולהשתנות על ידי המתכנת (כשמו כן הוא). על המשתנים חייבים להצהיר מראש ובכך מקצים זיכרון -נפח הזיכרון נמדד ב bytes

דוגמא להצהרה על משתנה:
int x;
המשתנה הוא **x**, מטיפוס **int** שלם

בהגדרת המשתנים חשוב (אין זו חובה אך מומלץ ביותר) להשתמש בשמות משמעותיים על מנת שהתוכנית שלנו תהיה קריאה ומובנית. נראה מספר דוגמאות: נניח כי התוכנית שלנו סוכמת את המספרים מ 1 עד 10, אזי נגדיר משתנה ושמו sum ברור כעת שמשתנה זה יכיל בסיום התוכנית את הסכום של המספרים שסוכמו. או לדוגמא - תוכנית הקולטת שמות סטודנטים וציונם, אזי נגדיר שני משתנים הראשון name לצורך שמירת שמו של הסטודנט ו grade -לצורך שמירת ציונו.

הערה: שם המשתנה מוגבל בדרך כלל עד 32 תווים. כך שאם נצהיר על משתנה עם 42 תווים המהדר יתעלם מעשרת התווים האחרונים.

הערה: הקומפיילר של c רגיש לשינויים בגודל האות כך שמשתנה ששמו name, Name, - NAME הם למעשה שלושה משתנים ולא אחד ולכן יש להקפיד על סגנון כתיבת המשתנים.

בלוק -סדרת פקודות המופיעות בתוך סוגריים מסולסלים {a1;a2; ….,an;} כאשר a1,a2,…,an פקודות בשפה.

מקום הגדרת המשתנים

ניתן להגדיר משתנים מחוץ לפונקציות או בתחילתו של בלוק כאשר מצהירים על משתנים. בבלוק ההצהרה חייבת להתבצע בתחילת הבלוק ולפני כל הוראה אחרת כלומר אם קיימת פקודה כמו **clrscr()** לפניי הצהרת המשתנים המהדר יתריע על קיומה של טעות .
כמו כן טווח ההכרה של משתנים אלו הוא בבלוק בלבד.

משתנים נחלקים לשני סוגים:
משתנים גלובליים ומשתנים לוקאליים.

משתנים גלובליים: משתנים המוצהרים מחוץ לפונקציה, מהווים משתנים גלובליים ביחס לפונקציה זו מכיוון שעדיין לא דיברנו על פונקציות ב c, -משתנים גלובליים לעת עתה בעבורנו הם משתנים שהצהרתם היא מחוץ ל main() כל הפונקציות שבאות אחרי ההצהרה מכירות את המשתנים הללו, כלומר משתנה המוצהר מחוץ לפונקציה והוא מוכר על ידי הפונקציות המוגדרות לאחר הצהרתו ועד סוף התוכנית.

משתנים לוקאליים: משתנים המוצהרים בתוך הפונקציה ואין פונקציות חיצוניות שמכירות אותם, כלומר "משך חייהם" הוא בתוך הפונקציה עצמה (שוב- כשנדון בפונקציות נראה את השימושים).

במקום להראות מימוש של משתנים דרך פונקציות נוכל להמחיש זאת על ידי שימוש בבלוקים.

התייחסותם של המשתנים כלפי פונקציות זהה לגבי בלוקים כלומר משתנה המוצהר מחוץ לבלוק מהווה משתנה גלובלי ביחס אליו, לעומת זאת משתנה המוצהר בתוך הבלוק מהווה משתנה לוקאלי

בס"ד

ביחס אליו.

סוגי משתנים

- **char** משתנה לאחסון תווים או סמלים , תופס בית אחד בזיכרון ,ניתן לאחסן בו ערכים מספריים שלמים מ -128 עד +127 תווים וסימנים הם כל האותיות, סימני פיסוק או סימונים אחרים שניתן להפיק ל ידי המחשב.

- **int** משתנה לאחסון מספר שלם תופס שני בתים בזיכרון ניתן לאחסן בו ערכים שלמים מ -32768 עד +32767.

- **float** משתנה של מספרים ממשיים עד 6 ספרות דיוק (מאוחסנים בשיטת נקודה צפה) המשנה תופס ארבעה בתים בזיכרון

- **double** משמש לאחסון ערכים ממשים בתחומים גדולים יותר, עד 10 ספרות דיוק ,תופס שמונה בתים בזיכרון.

- **long** ניתן לאחסן בו ערכים שלמים בטווח גדול יותר מ int תופס ארבעה בתים בזיכרון.

- **sing/unsing** כל משתנה מספרי ניתן להגדיר כ sing עם סימן +/- או unsing ללא סימן
כאשר sing מציין אחסון של מספרים שלילים וחיוביים ואילו unsing מציין אחסונם של מספריים אי שליליים ,כתוצאה מהגדרת משתנה כ unsing טווח החיוביים שלו גדל.
לדוגמא sing char ניתן לאחסן ערכים מהטווח (-128 +127) לעומת זאת אם הגדרנו את המשתנה כ unsing char ניתן לאחסן בו ערכים מהטווח.(0 255)
ברירת המחדל מגדירה משתנה אוטומטית כ sing

שמות חוקיים למשתנים

יש להקנות שם ייחודי לכל משתנה ובכך אנו למעשה מאפשרים לאחסן בתא מסוים בזיכרון ערך וגם נוכל לפנות אל תא זה על מנת לקרוא ממנו את הערך.
לא ניתן לקרוא למשתנה בכל אופן שעולה על רוחנו אלא ישנם חוקים לשמות משתנים הללו חייבים להתחיל באות או בקו תחתי (אך מומלץ להשתמש האותיות בלבד) כמו כן לא נוכל להקנות שמות שמוגדרים כמילים שמורות כמו else if int וכו.
שאר תווי שם המשתנה יכולים להיות מספרים, אותיות וקו תחתי אך יש להקפיד לא לכלול סימנים נוספים כגון נקודה, פסיק ורווח.

טבלת הטיפוסים

תחום המספרים	מספר בתים	טיפוס
128 - 127+	1	char, signed char
0 - 255+	1	unsigned char
32767+ 32768-	2	int ,sign int
65535+ 0	2	unsign int
2147483647+ 2147483648-	4	long,long int,signed long int
4294967295+ 0	4	unsigned long,unsigned long int
3.4e+38 3.4e-38	4	float ספרות דיוק 7
1.7e+308 1.7e+308	8	doubleספרות דיוק 15
3.4e+4932 3.4e4932	10	long doubleספרות דיוק 19

הערה: בהצהרה על משתנה יש לרשום את טיפוס המשתנה ואחר כך את שמו כאשר חייב להיות רווח אחד לפחות בניהם (ניתן לכתוב מספר רווחים אך אין בכך הבדל).
כאשר ישנם מספר משתנים בעלי אותו טיפוס ניתן להצהיר עליהם באשר נכתוב בראשית את סוג הטיפוס ואחר כך כשפסיקים מפרידים נכתוב את שמות המשתנים לדוגמא:
int sum, age, num_of_students.
אם מעוניים ששמות המשתנים יהיו מטיפוס sing אין צורך להצהיר בתוספת המילה השמורה sing מכיוון שזהו ברירת המחדל. ולכן אם נרצה שהמשתנה יהי unsing יש להצהיר על כך.

השמת ערכים ואתחול משתנים

כאשר מצהירים על משתנה חדש ערכו לא ידוע. הוא יכול להכיל ערך שנקבע לו בתוכנית הקודמת או אפס או כל ערך אחר ולכן כדי לבצע השמה של ערך במשתנה משתמשים באופרטור השוויון. לדוגמא - אם נרצה לתת למשתנה ששמו grade את הערך 100 נרשום:
;grade = 100

דוגמא 1:

```
#include <stdio.h>

void main()
{
   int a, b, c;
   a = 10;
   b = 20;
   c = (a+b)/2;
   printf("the average of a and b is: %d", c);
}
```

מפני שהפקודה תקרא את שם המשתנה ולא תציג את ערכו, הוחלט על תבניות המרה המאפשרות

בס"ד

הדפסת משתנים והן:

%d להצגת integer
%f להצגת float
%ld להצגת long
%c להצגת char
%s להצגת string
%lf להצגת double

יש לציין שניתן לקבוע את רוחב השדה, מס' הספרות אחרי הנקודה העשרונית, ישור לימין, לשמאל וכו'
עד עתה כאשר רצינו להדפיס מספר השתמשנו ב %d נוסיף את הפרמטרים לקביעת רוחב השדה, ישור ומספר ספרות אחרי הנקודה כדלהלן: כאשר נהיה מעוניינים להדפיס מספר ברוחב 3 ספרות וישור לשמאל נרשום %d3 אם נהיה מעוניינים בישור לימין (רוחב של 3 ספרות) נרשום %d3 - כשאר מדובר במספרים ממשיים אזי כדי להדפיס לדוגמא מספר ממשי ברוחב של 12 תווים מתוכם 2 תווים לאחר הנקודה העשרונית נרשום כך %lf12.2

תבניות אלה נכתבות בין המרכאות של הפונקציה **printf()** אך שמות המשתנים נכתבים מחוץ למרכאות ומופרדים על ידי פסיקים כדלהלן:

נצהיר על שני משתנים :
```
int a = 1;
int b = 2;
```
כעת נדפיס אותם :
```
printf (“Print: a = %d , b = %d”, a,b);
```

הערה: ניתן להצהיר על מספר משתנים מאותו טיפוס כך:
int/float… a,b,c;
כלומר כתיבת שם הטיפוס ואחר כך כתיבת שמות המשתנים ובניהם פסיקים לדוגמא:
int a, b, c;
המחשת ההבדל בין משתנים לוקאליים לגלובלים.

דוגמא 2:

```
#include <stdio.h>

void main()
{
   int a = 2;
   int b = 3;
   int sum = a+b;

   printf ("The value of sum is: %d \n\r", sum);
   printf("The value of a+b is: %d \n\r", a+b);

   {
      int local = 4;
      printf("The value of local is: %d \n\r", local);
      printf("The value of sum is: %d \n\r", sum);
```

```
        printf("sum is a global variable ");
    }
    printf("But here the variable local is not known");
}
```

Casting זוהי המרה יזומה ,כלומר כשאנו מעוניינים לשנות טיפוס יש לרשום את הטיפוס אליו מעוניינים לשנות בסוגריים בדוגמא שלנו, (int) כמובן שלא ניתן לבצע המרה לטיפוס מסוג אחד לסוג אחר באופן הנ"ל:

```
int integer = 8;
char c = 'c';
integer = (float)c;
```

וזאת מכיוון שהמשתנה integer הוא מטיפוס שלם ולא ניתן להציב בו ערך אחר משלם.
אומנם ניתן להציב את הערך המומר למשתנה אחר מאותו טיפוס.

לדוגמא:

```
float f ;
f = (float)c;
```

כאשר נדון באופרטורים נפרט יותר על המרת טיפוסים.

דוגמא 3:

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

void main()
{
    int a = 2;
    float b = 3.0;

    printf ("The value of a is: %d\n\r", a);
    printf("The value of a+b is: %f\n\r", a+b);
    printf ("The value of b/a is: %f\n\r", b/a);
    printf ("The integer value of b/a is: %f\n\r", (int)(b/a));
}
```

כעת נרצה לקלוט ערכי משתנים מהמשתמש , הפונקציה לקבלת קלט מהמשתמש היא **scanf()** אשר בדומה לפונקציה **printf()** היא מקבלת בתוך המירכאות תבניות המרה , ושמות משתנים עם הסימן **&** לפני (נדון בסימן זה בהרחבה בפרק על מצביעים).

בס"ד

לדוגמא:

```
int a;
float b;
printf("Enter values for a and b");
scanf("%d %f", &a, &b);
```

דוגמא 4:

```
#include <stdio.h>

void main()
{
    int a;
    float b;

    printf("Enter values for a and b: \n\r");
    scanf("%d %f",&a,&b);
    printf ("The value of a is: %d\n\r", a);
    printf("The value of b is: %f\n\r", b);
    printf("Change the value of a \n\r");
    scanf("%d", &a);
    printf("The new value of a is:%d", a);
}
```

הגדלה עצמית והקטנה עצמית של משתנים

x++ פירושה הגדל את ערך המשתנה x ב -1, ועדכן את x (x = x+1)
x-- פירושה הקטן את ערך המשתנה x ב -1, ועדכן את x (x = x-1)
אך קיימות גם הפעולות x++ ו x--

נסביר את ההבדל בין x++ ל ++x על ידי דוגמא:
y=x++ פירושה y מקבל את הערך הראשוני של x ואח"כ x גדל ב1
y=++x פירושה x גדל ב 1 ואח"כ y מקבל את ערכו של x

לסיכום כאשר נשתמש באופרטור לפני המשתנה , המשתנה יקודם תחילה, ורק אחר כך נוכל להשתמש בערכו המעודכן. וכאשר האופרטור יופיע לאחר המשתנה ,קודם נשתמש בערכו הקודם של המשתנה ורק אחר כך ערכו יקודם או יוקטן.

בס"ד

דוגמאות:

1.

```
int x,y;
x = 10;
y = x++;
```

בסיום קטע הקוד נקבל:

x = ?, y = ?

2.

```
int x,y;
x = 10;
y = x--;
```

בסיום קטע הקוד נקבל:

x =?, y = ?

3.

```
int x, y, z;
z = 10;
x = z++;
y = ++z;
```

בסיום קטע הקוד נקבל:

x = 10, y = 12

דוגמא 5:

```
#include <stdio.h>

void main()
{
    int a = 10;
    int b = 5;
    int result;

    result = a + b;
    printf("The value of result is: %d\n\r",result);
    result = ++a + b;
    printf("The value of result is: %d\n\r",result);
    result = a + b--;
    printf("The value of result is: %d\n\r",result);
    result = a-- + b;
    printf("The value of result is: %d\n\r",result);
    result = a + --b;
    printf("The value of result is: %d\n\r",result);
    result = ++a + ++b;
    printf("The value of result is: %d\n\r",result);
}
```

בס"ד

מקרו

מקרו מהווה קטע קוד שאשר במהלך התוכנית אם נקרא למקרו, קטע הקוד שהוגדר כמאקרו "יושתל" בקריאה למקרו.
מגדירים מקרו בהוראה # define
תבנית למקרו:
#define macro definition
לדוגמא:
#define LENG 10

לפני הידור התוכנית, המהדר יחליף את LENG בכל מקום בתוכנית בערך 10 ואז יהדר את התוכנית.
נראה שימוש לכך כשנדון במערכים

בס"ד

אופרנדים ואופרטורים

אופרנדים

אופרנדים הם משתנים או קבועים. בפרק הקודם דנו במשתנים, בפרק זה נעסוק גם בקבועים.

קבועים (constants) - משתנים שמאותחלים לערכים קבועים וידועים מראש שאינם משתנים במהלך ביצוע התוכנית. כל קבוע מיוחס לסוג משתנה מסוים, לדוגמא pi הוא קבוע מסוג double וערכו ידוע לכל (3.1415) . אם נרצה להשתמש בתוכניתנו ב- pi לא נרצה לאפשר להציב ערך אחר ל- pi (כלומר לא נאפשר לעצמנו לשנות את ערכו של pi). כדי לעשות זאת אנו מגדירים את המשתנה כקבוע.
על מנת להגדיר את המשתנה כקבוע מוסיפים את המילה השמורה const לפני ההצהרה על סוג הטיפוס.

דוגמאות:
const double pi = 3.1415;
const int age = 65;
const long phone = 6543232;
אפשר לאלץ כל קבוע להיות int, double, float וכו' על ידי הוספת סיומת מתאימה, והן:
אם נרצה שהקבוע יהיה מסוג long יש להוסיף את הסיומת L.
אם נרצה שהקבוע יהיה מסוגfloat יש להוסיף את הסיומת F.
אם נרצה שהקבוע יהיה מסוג double יש להוסיף את הסיומת D.
אם נרצה שהקבוע יהיה מסוג unsing יש להוסיף את הסיומת U.

דוגמאות:
;

age = 65L age = 65L
pi = 3.1415F;
בדוגמא הראשונה המרנו את טיפוס המשתנה age מ int ל long .
בדוגמא השנייה המרנו את טיפוס המשתנה pi מ-double ל- float .

הערה: לא כל המהדרים תומכים בסיומות אלה, התמיכה היא רק במהדרים החדשים.

אפשר כמו כן גם לבצע השמה של משתנים מטיפוסים שונים אך חשוב לשים לב כשעושים זאת כי במקרים מסוימים אנו עלולים לאבד מידע. לדוגמא אם נבצע השמה של משתנה מטיפוס float ל-int נאבד את החלק הלא שלם של המספר.

אופרטורים (operators)

אופרטור השמה - מיוצג על ידי תו = כאשר ערך הביטוי מצד ימין מוצב במשתנה מצד שמאל:
int a = 5;
int b = a;
תחילה המשתנה a יקבל את הערך 5 ואח"כ המשתנה b יקבל את ערכו של משתנה a כלומר ערכו של b לאחר משפט זה יהיה 5.

אופרטורים חשבוניים - מאפשרים לבצע פעולות חשבוניות כמו חיבור, חיסור, חילוק, ומודולו (שארית):
a+b חיבור המשתנים a ו- b .
a-b חיסור המשתנה a ב- b .

a*b מכפלת המשתנים a ו- b .
a%b שארית של המשתנה a בחלוקה ב- b.

	לפני	לפני	לפני	אחרי	אחרי	אחרי
טיפוס	**a**	**b**	**C**	**a**	**b**	**c**
a = b + c	1	2	3	5	2	3
a = b - c	1	2	3	-1	2	3
a = b * c	1	2	3	6	2	3
a = b / c	1	4	2	2	4	2
a = b % c	1	9	5	4	9	5

ניתן כמו כן לרשום מספר פעולות חשבוניות בביטוי מורכב אחד, לדוגמא:
Price = ((old_price * 80)%100)+5;
שמשמעו המחיר לאחר הנחה של 20 אחוזים ועוד תוספת של 5 שקלים (בעבור שירות).

דוגמא 1:

```
#include <stdio.h>

void main()
{
   int a, b, sum, diff, mult, rem, quot;

   a = 10;
   b = 8;
   sum = a + b;
   diff = a - b;
   mult = a * b;
   rem = a % b;
   quot = a / b;

   printf ("sum is: %d \n", sum);
   printf ("difference is: %d \n", diff);
   printf ("multiplication is: %d \n", mult);
   printf ("remainder is: %d \n", rem);
   printf ("quotient is: %d \n", quot);
}
```

אופרטור השמה עצמית - ניתן לקצר את כתיבת הפקודה כאשר מעוניינים להשתמש בערכו של המשתנה, לבצע עליו פעולה ולאחסן את התוצאה במשתנה עצמו:
נמחיש זאת על ידי דוגמא x *= 5 :שקול לx = x*5
x /= 5 שקול לx = x/5
x %= 5 שקול לx = x%5

בס"ד

x -= 5 שקול לx = x-5
x += 5 שקול לx = x+5
x *= 5 שקול ל x = x*5

יש להקפיד שלא יהיו רווחים בין האופרטור הרצוי לאופרטור ההשמה.(=)

דוגמא 2:

```
#include <stdio.h>

void main()
{
    int a = 5;
    int b = 10;

    a += b;
    printf("The value of a is: %d\n\r", a);
    a /= b;
    printf("The value of a is: %d\n\r", a);
    a -= a;
    printf("The value of a is: %d\n\r", a);
    b *= ++a;
    printf("The value of b is: %d\n\r", b);
    b %= (a+5);
    printf("The value of b is: %d", b);
}
```

אופרטורי יחס

< יחס קטן (מכיוון שסדר ביצוע הפעולות הוא משמאל לימין)
> יחס גדול.
== יחס שיווין.
=< יחס גדול או שווה.
=> יחס קטן או שווה.

אופרטורים לוגיים - לערכים בוליאניים (אמת או שקר) אין בשפת c משתנה מיוחד ומשתמשים למטרה זו במשתנים שלמים כאשר ערך 1 מייצג אמת וערך 0 מייצג שקר.

&& פירושו וגם.
|| פירושו או.
=! פירושו שונה.
! פירושו לא.
כך לדוגמא:
int good_grade = (grade > 80);
אם ערכו של המשתנה grade יהיה מעל 80 אזי ערך המשתנה good_grade יהיה 1 אחרת 0.

דוגמאות:
b = 5

בס"ד

a = 10
(a > b) ערכו של ביטוי זה אמת.
((a-5) == b) ערכו של ביטוי זה אמת.

סדר קדימויות (מעדיפות עליונה לתחתונה):

```
++ , -- , !
* , % , /
+ , -
< , =< , => , >
== , !=
&&
||
= , =+ , =- , =/ , =*
```

בס"ד

פלט – קלט

הספרייה stdio

בפרקים קודמים הכרנו באופן שטחי את פונקצית הקלט scanf ופונקצית הפלט - printf עבור קלט ופלט סטדנדרטים מתוך הספרייה stdio בפרק זה נכיר עוד מספר פונקציות של הספרייה stdio ובניהן:
printf ו scan -עבור קלט ופלט מובנה.
puts ו gets -עבור קלט ופלט של מחרוזות.
putchar ו getchar -עבור תווים בודדים.
כלומר בפרק זה נראה פונקציות קלט פלט אופייניות לטיפוסים ספציפיים בלבד.

פלט של תו בודד - הפונקציה putchar()

הפונקציה putchar() מאפשרת לנו הדפסה של תו בודד למסך.
נמחיש זאת על ידי דוגמא:
putchar('H');
פקודה זו תדפיס את התו H על המסך.
ניתן כמו כן להכניס ערך תווי לתוך משתנה ולהזין את שם המשתנה לתוך פקודת putchar ובכך יודפס ערכו של המשתנה, לדוגמא:
char ch = 'a';
putchar(ch);

תבנית של הפונקציה putchar()
putchar (char);
נכתוב תו בין גרשיים בודדים לדוגמא 'H': אך אם נרצה להדפיס את ערכו של משתנה תווי נרשום בין הסוגריים העגולים של הפונקציה putchar ללא גרשיים בודדים או ערך asci גם ללא גרשיים.

קלט של תו בודד - הפונקציה getchar()

הפונקציה getchar() מאפשרת קליטה של תו בודד מיחידת הקלט הסטנדרטי (מקלדת).
בקליטת התו נרצה לשמור על ערכו ולכן נדאג שערך המוחזר על ידי הפונקציה getchar ישמר במשתנה תווי.
לדוגמא:
char c;
c = getchar();
כך שאם המשתמש הקיש H אז המשתנה c יאחסן את הערך H

תבנית של הפונקציה getchar()
char = getchar();
אנו נשתמש בפקודה זו רבות מכיוון שאם אנו מעוניינים לראות את הפלט של התוכנית אנו נרשום את הפקודה בסיום התוכנית ובעקבות כך התוכנית תיעצר ותחכה לקלט מהמשתמש. התוכנית לא תתקדם עד אשר היא תקבל קלט מהמשתמש ובהזדמנות זו המשתמש יכול לקרוא את הפלט של התוכנית וכשיסיים יקיש מקש כלשהו. את התו שהוקש לא נשמור במשתנה (כי ערכו לא רלוונטי לנו)
.

פלט מובנה - הפונקציה printf()

פונקציה זו מאפשרת להדפיס מספר משתנים שונים על ידי אותה פקודה (כלומר במשפט אחד).
מספר הערכים שאנו מעוניינים להדפיס יכול להשתנות, הפקודה printf() תומכת בשינוי של מבנה

הפלט בהתאם לצרכינו.
ניתן גם להדפיס מחרוזות ללא שימוש במשתנים כלשהם. במקרה כזה נרשום את המחרוזת בין מרכאות בתוך הסוגריים העגולים של הפקודה printf() בלבד.
כזכור ניתן להוסיף לפקודת תווים שונים בהתאם לצרכינו - לדוגמא, הוספת "\n" מאפשרת מעבר לשורה חדשה וכו.'

תבנית של הפקודה: printf()
printf (“ ….”, first_v, second_v, …, last_v);
בין המרכאות של הפקודה printf() יש לרשום את תבנית ההמרה המתאימה לכל משתנה.

דוגמא:

```
int a = 3;
float b = 5.0;
long c = 8.00;
prinf(“Hello\n”);
printf(“The sum of %d %f is %ld”,a ,b, c);
```

קלט מובנה - הפונקציה scanf()

פונקציה זו מאפשרת קליטה מובנת של ערכים. בעזרת פונקציה זו ניתן לקלוט מספר ערכים מטיפוסים שונים
כמו שפקודת printf() מאפשרת להדפיס מספר רב של משתנים מטיפוסים שונים .

מספר הערכים שאנו מעוניינים לקלוט יכול להשתנות, הפקודה scanf() תומכת בשינוי של מבנה הקלט בהתאם לצרכינו.

תבנית של הפקודה: scanf()
scanf("……", &addres_of_first_v, &address_of_second_v,…,&adderss_of_last_v);
בין המרכאות של הפקודה scanf() יש לרשום את תבניות ההמרה של המשתנים, כלומר אם רוצים לקלוט משתנה מטיפוס שלם יש להוסיף בין המרכאות את תבנית ההמרה %d ולפני שמו של המשתנה יש להוסיף את אופרטור הכתובת (&) וכך לכל המשתנים. יש להקפיד על סדר כתיבת המשתנים ותבניות ההמרה.
בעצם על ידי כתיבת אופרטור הכתובת אנו דואגים להכניס את הערכים לכתובות המתאימות של המשתנים בהתאמה.

הערה :אם נשכח את האופרטור הכתובת (&) אנו עלולים לגרום לקריסת התוכנית.

דוגמא:

```
long num_of_course;
float grade;
long id;
printf ("Enter your num of course id number and grade \n");
scanf("%ld %ld %f", &num_of_course, &id, &grade);
```

נסכם:
- getchar() פונקצית קלט של תו בודד. אין להזין ערכים בין הסוגריים של הפונקציה, את הערך

המוחזר (כלומר שנקלט מהמשתמש) נשמור במשתנה.
- putchar() פונקצית פלט של תו בודד.
יש להזין בין הסוגריים של הפונקציה תו עם גרשיים בודדים או שם של משתנה תווי או ערך asci
- printf() פונקצית פלט של מספר משתנים מטיפוסים שונים ומחרוזות.
- scanf() פונקצית קלט של מספר משתנים מטיפוסים שונים.

ניקוי החוצץ - הפונקציה - flushall()

fflush(stdin);

חוצץ הינו קטע זיכרון המיועד לאחסנה זמנית של קלט לפני שהתוכנית מטפלת בו ולאחסנה זמנית של פלט שהתוכנית עורכת, בדרכם אל המסך או המדפסת.
כלומר, כל פעולות הקלט והפלט מתבצעות דרך חוצץ ולכן עלולים להימצא נתונים לא רצויים בחוצץ אשר עלולים לשבש את ריצת התוכנית וכדי למנוע זאת מנקים את החוצץ.
דוגמא:

```
char ch;
int num;
printf(“Enter your number: “);
scanf(“%d”, &num);
flushall();
printf(“enter your char:”);
ch = getchar();
```

הערה: אם לא נרשום את הפקודה flushall() התוכנית לא תחכה לקליטת תו מהמשתמש וב ch- יהיה ערך "זבלי."

דוגמא 1:

```
#include <stdio.h>

void main()
{
    char ch;
    int a;

    printf("Enter value to ch ");
    ch = getchar();
    printf("\n\rThe value of ch is: ");
    putchar(ch);
    a = (int)ch;
    printf("\n\rThe value of a is: ");
    putchar(a);
    printf("\n\rChange the value of ch ");
    flushall();
    scanf("%c",&ch);
    printf("\n\rThe asci value of ch is: %d",ch);
}
```

בס"ד

תנאים וללואות

תנאי בקרה

מטרתם לאפשר לנו לשנות את רצף התוכנית, לדלג על פקודות ולבצע פקודות מספר פעמים .
נחלק את משפטי הבקרה לשלושה סוגים:

פקודות לביצוע מותנה:
הפקודות if ו- else
הפקודות switch ו- case
האופרטור ?

פקודות לביצוע לולאות:
הפקודות for
הפקודה while
הפקודה do while
(ופקודות עזר break ו- continue)

המשך במקום אחר בתוכנית:
goto

משפט if

משפט בקרה המאפשר לנו ביצוע מותנה של הוראות או פקודות כאשר תנאי מסוים נבדק ומבוצע בהתאם לערכו של התנאי: אמת - המיוצג על ידי כל מספר השונה מ- 0, ושקר - המיוצג על ידי הספרה 0.
משפט if מאפשר לנו לסעף את תוכניתנו על ידי תנאי לוגי באופן הבא:

תבנית של הפקודה if:

```
if (term)
{
   expression a;
}
else
{
   expression b;
}
```

כלומר אם התנאי בתוך הסוגריים של משפט ה if מתקיים ערך term הוא "אמת" כאשר ערך התנאי שונה מאפס ,בצע את הפקודות המופיעות בין הסוגריים המסולסלים (expression a) אם התנאי לא מתקיים בצע את סדרת הפקודות המופיעות בין הסוגריים המסולסלים השניים (expression b)

הערה:
אפשר גם ללא סוגריים מסולסלים אבל אז יתבצע רק המשפט הראשון שלאחר התנאי , בהנחה שהתנאי אמת.

דוגמא 1:

```
#include <stdio.h>

void main()
{
   int a, b;

   printf("Enter a value for a \n\r");
   a = (int)getchar();
   printf("Enter a value for b \n\r");
   flushall();
   b = (int)getchar();

   if(a > b)
     printf ("a is bigger then b");
   else
      printf ("b is bigger or equal to a");
}
```

הערה:
משפט if יכול להופיע גם בתוך משפט - else לדוגמא, בדוגמא זו, אם התנאי הראשון נכשל אזי המספר השני גדול או שווה למספר הראשון. כעת נשאל לאחר משפט else למעשה - בתוך פסקת else אם b גדול מ a אם לא - ברור לנו כי a ו b -שווים. נראה זאת בדוגמא הבאה.

דוגמא 2:

```
#include <stdio.h>

void main()
{
   int a, b;

   printf("Enter values for a and b\n\r");
   scanf("%d %d",&a, &b);

   if (a > b)
      printf ("a is bigger then b");
   else
      if (b > a)
         printf ("b is bigger then a");
      else
         printf("a and b are equal");
}
```

אם מעוניינים שמתוך סדרת תנאים יבוצע רק אחד מהם נוכל להשתמש בתבנית הבאה.

תבנית לשימוש בפקודת: if else

```
If (exp)
{
  . . .
}
else if(exp)
{
  . . .
}
else if(exp)
{
  . . .
}
else
{
  . . .
}
```

הערה:
התנאי הראשון יהיה if והאחרון יהיה else
נבדק התנאי הראשון - אם הוא מצליח (ערכו אמת) מבוצע הבלוק המתאים לו. אם השני נכשל - עוברים ל if -הבא, אם הוא מצליח מבוצע הבלוק המתאים, אחרת עוברים ל if -הבא וכך הלאה.

הערה:
אם היינו משתמשים בפקודת if רגילה, כלומר יוצרים רצף של פקודות if ההבדל העיקרי הוא שגם אם פקודה if מצליחה הבלוק המתאים מבוצע אך ממשיכים לבדוק את שאר התנאים ואם גם בהם תנאי נוסף מתקיים מבצעים גם את הבלוק המתאים לו. זאת בניגוד לפקודת - if else שם אם תנאי אחד מתקיים שאר התנאים לא נבדקים ויוצאים מקטע קוד זה של בדיקות פקודות ה if

דוגמא 3:

```
#include <stdio.h>

void main()
{
   int a

   printf("Enter a number between 0-100: ");
   scanf("%d", &a);
   if (a < 0)
      printf("The number is not at the target");
   else if (a < 10)
      printf("The number is between 0-10");
   else if (a < 20)
```

```
      printf("The number is between 10-20");
   else if (a < 30)
      printf("The number is between 20-30");
   else if (a < 30)
      printf("The number is between 30-40");
   else if (a < 40)
      printf("The number is between 30-40");
   else if (a < 50)
      printf("The number is between 40-50");
   else if (a < 60)
      printf("The number is between 50-60");
   else if (a < 70)
      printf("The number is between 60-70");
   else if (a < 80)
      printf("The number is between 70-80");
   else if (a < 90)
      printf("The number is between 80-90");
   else if (a < 100)
      printf("The number is between 90-100");
   else
      printf("The number is bigger then 100");
}
```

בפקודת ה if-נשתמש באופרטורים:

> , < , >= , <= , == ,!= , !

תנאים מורכבים:

עד כה הדגמנו תנאים פשוטים כמו: קטן מ, שווה ל, שונה מ- וכו'. כעת נראה איך ניתן לכתוב תנאים מורכבים על ידי שימוש באופרטורים לוגיים : וגם (&&) או (||)
נמחיש על ידי דוגמא:

```
 int a, b;
printf("Enter 2 numbers: \n");
scanf("%d %d", &a, &b);
if (a > b && a == 5)
    printf("a > b and a == 5\n");
if (b > 7 && b < a)
    printf(" b > 7 and b < a\n");
if (!a || !b)
    printf("a or b is zero");
```

סדר ביצוע פעולות בביטויים לוגיים:
ביטוי לוגי נבדק תמיד משמאל לימין עד אשר ניתן יהיה לקבוע בוודאות את ערך הביטוי - אמת או שקר. לדוגמא:

נניח שערך הביטוי x > y הוא שקר ורשמנו If (x > y && y++)
-ברור שערך הביטוי במשפט ה if -יהיה שקר, אך מאחר והביטוי נבדק משמאל לימין ומדובר בביטוי "וגם" (כל חלקיו צריכים להיות אמת על מנת שכולו יקבל ערך אמת) - הבדיקה תיפסק לאחר החלק הראשון ברגע שיסתבר שהוא שקר, ולכן החלק השני כלל לא ייבדק והמשתנה y לא יקודם באחד.

משפט swich

בדוגמא האחרונה הראנו בחירה מרובה על ידי משפט if מורכב. כעת נראה כיצד ניתן לפשט אותו ולאפשר דרך נוחה וברורה יותר לביצוע הפעולה באופן הבא:

-constant1 קבוע ראשון
..
..
- constantn הקבוע ה n

```
switch(exp or variable)
{
   case constant1: command; break;
   case constant2: command; break;
   case constant3: command;break;


   case constantn: command;break;
   default: command;
}
```

switch מקבל משתנה או ביטוי ומחשב את ערכו. הערך המתקבל הוא קבוע (ערך מספרי או אלפבתי) והפקודה שמתאימה לקבוע שחושב מתבצעת, כלומר התוכנית פונה אל שורת ה case - המתאימה ומבצעת את הפקודות הכתובות אחריה עד שהיא מגיעה לפקודת break
מטרת הפקודה break היא למנוע את המשך ביצוע הפקודות הבאות ב-case -ים שבאים אחרי.
אם הערך המתקבל ממשפט ה switch -אינו תואם לאף אחד מה-case -ים מתבצעות ההוראות שאחרי פקודת ה defaut-

הערה:
הפקודה default לא חייבת להופיע אך במקרה זה אם הקבוע שחושב ב switch אינו תואם את אחד ה -case -ים לא יבוצע דבר.

דוגמא 4:

```
#include <stdio.h>
void main()
{
   char ch;
   printf("Enter a char (between a-e): ");
   scanf("%c", &ch);

   switch(ch)
   {
      case 'a': printf("You have printed a");break;
      case 'b': printf("You have printed b");break;
      case 'c': printf("You have printed c");break;
      case 'd': printf("You have printed d");break;
      case 'e': printf("You have printed e");break;
      default: printf("Your char is not between a-e");
   }
}
```

וללא שימוש ב break נקבל:

דוגמא 5:

```
#include <stdio.h>

void main()
{
   char ch;
   printf("Enter a char (between a-e): ");
   scanf("%c", &ch);

   switch(ch)
   {
      case 'a': printf("You have printed a");
      case 'b': printf("You have printed b");
      case 'c': printf("You have printed c");
      case 'd': printf("You have printed d");
      case 'e': printf("You have printed e");
      default: printf("Your char is not between a-e");
   }
}
```

בס"ד

הביטוי המותנה?

התבנית של?

condition? exp1: exp2;

לפני ה- ? ישנו תנאי ובאם התנאי מתקיים יש לבצע את הביטוי הראשון. אחרת (התנאי אינו מתקיים) יש לבצע את הביטוי השני (המופיע מיד לאחר הנקודותיים :)
כמו כן ניתן להציב את הערך המתקבל במשתנה או לצורך הדפסה

בעצם ניתן לתרגם ביטוי זה למשפט if כך:

```
if (condition)
   exp1;
else
   exp2;
```

דוגמא:

```
int x, y, z;
if (x > y)
      z = x;
  else
      z = y;
```

על פי הביטוי המותנה ? נרשום כך:

```
  int x, y, z;
  z = (x >y) ? x : y;
```

כך המשתנה z
יקבל את הערך המקסימלי מבין המשתנים
x, y

דוגמא 6:

```
#include <stdio.h>

void main()
{
   char ch;

   printf ("Enter a latter (a-z): ");
   scanf("%c", &ch);
   ch = ('a' <= ch && ch <= 'z')? (ch+'A'-'a'):ch;
   printf("The result is: %c", ch);
}
```

בס"ד

לולאת for

לולאות מאפשרות לבצע הוראה או בלוק הוראות בתוכנית מספר פעמים בהתאם לצרכינו.
כאשר אנו מעוניינים כי בלוק של פקודות או פקודה בודדת יתבצעו מספר פעמים אנו משתמשים בלולאת for
לולאת for נקראת גם לולאת אינקדס.
הלולאה בנויה משלושה חלקים:

1- אתחול משתנים.
2- תנאי ביצוע.
3- קידום משתנים.

מבנה הלולאה:

```
for (initialization ; boolean condition ; increment)
{
   . . .
}
```

שלושת חלקי הלולאה מופרדים ביניהם על ידי ; (נקודה פסיק).
בחלק הראשון מתבצע אתחול משתנים, בחלק השני נבדק תנאי הלולאה - כאשר אם ערכו הבוליאני של תנאי זה אמת יבוצע גוף הלולאה ואם לא נצא מהלולאה ובחלק השלישי (המבוצע **לאחר ביצוע גוף הלולאה**) מבוצע קידום משתנים.
יש לציין כי האתחול מתבצע רק פעם אחת בכניסה ללולאה, אח"כ נבדק תנאי הלולאה ואם ערכו אמת מתבצע גוף הלולאה, אח"כ קידום משתנים ושוב בדיקת תנאי הלולאה וחוזר חלילה. הרוטינה תתבצע עד אשר לא יתקיים תנאי הלולאה ואז נמשיך בתוכנית לאחר לולאת ה for -

הערה:
אפשר לוותר על הסוגריים אם גוף הלולאה מורכב משורת קוד אחת.

דוגמא 7:

```
#include <stdio.h>

void main()
{
   int times;
   printf("Printing the numbers form 10 to 1: \n\r");
   for (times = 10 ; times > 0 ; times--)
      printf ("%d ", times);
}
```

השמטת חלקים בלולאה:
לעיתים נרצה לוותר על אחד ממרכיבי הלולאה.
כל אחד משלושת חלקי הלולאה (אתחול, תנאי לולאה, קידום) יכול להיות ריק.
לדוגמא:

```
int a = 5;
for ( ; a < 5; a++)
{
   . . .
}
```

בדוגמא זו דילגנו על אתחול המשתנה מאחר ואיתחלנו אותו מראש.

```
int a;
for (a = 5; a<5 ; )
{
   . . .
a++;
}
```

בדוגמא זו השמטנו את קידום המשתנה, הקידום נעשה בגוף הלולאה.

לא ניתן להשמיט את החלק של תנאי הלולאה כי אז נקבל לולאה אינסופית למרות שניתן להתגבר על הבעיה באמצעות break אך צורה זו פחות מקובלת

דוגמא 7.1:

```
#include <stdio.h>

void main()
{
    printf("Printing the numbers form 10 to 1: \n\r");

    int times = 10;

    for ( ; times > 0 ; times--)

        printf ("%d ", times);
}
```

אין הכרח שקידום המשתנה יהיה באחד בכל איטרציה. לדוגמא:

```
int a;
for (a = 40; a >=0 ; a +=2)
    printf("%d ",a);
```

בדוגמא זו אנו מדפיסים את המספרים הזוגיים מ 40 עד ל 0.
קידום המשתנה הוא ב- 2.
כמו כן אין הגבלה למשתנה אחד, כלומר אפשר לבצע אתחול של מספר משתנים, בדיקת תנאי לוגי על מספר משתנים וקידום של מספר משתנים יחד. לדוגמא:

```
int counter ;
char ch ;
for (ch = 'a', counter = 1; counter <= 26 ; ch++, counter++)
    printf("%d. %c ",counter, ch);
```

קטע קוד זה ידפיס את האותיות האנגליות מ a -עד z כאשר בכל שורה יופיע אינדקס האות, נקודה מפרידה ואחריה האות עצמה.
רואים כי כאשר אנו עובדים עם מספר משתנים יש להקפיד על פסיקים בניהם.

קינון לולאות:
ניתן לבצע קינון לולאות, כלומר לולאה בתוך לולאה. יש להתייחס אל הלולאה הפנימית כמו אל פקודת ביצוע שכאשר היא מסתיימת עוברים לחלקה השלישי של לולאת ה for -החיצונית וקידום המשתנים (קידום המשתנים משמע שינוי ערכם אם עלי ידי הגדלה עצמית או הקטנה עצמית כמו בדוגמא האחרונה) ואח"כ שוב ביצוע גוף הלולאה החיצונית (שהיא בעצמה גם לולאה) וכך הלאה..

דוגמא 7.2:

```
#include <stdio.h>

void main()
{
    double i;
    double result = 1;

    printf ("Factorial function:\n\r");
    for (i = 0; i < 10; i++)
    {
        if (i != 0)
       result *= i;
        printf ("%lf\n\r", result);
    }
}
```

דוגמא 8:

```
#include <stdio.h>

void main()
{
    printf("Printing the multiplication table: \n\r");
    int i,j;

    for (i = 1 ; i <= 10 ; i++)
    {
      printf("\n\r");
      for (j = 1 ; j <= 10 ; j++)
        printf ("%2d ", i*j);
      }
}
```

דוגמא 8.1:

```
#include <stdio.h>

void main()
{
    int i, j;

    for (i = 0; i < 6; i++)
    {
      printf ("\n\r");
      for (j = 0; j <= i ; j++)
        putchar ('*');
    }
}
```

לולאת while

כמו לולאת for , לולאת while משמשת לחזרה על בלוק של פקודות (או פקודה בודדת) כאשר תנאי הלולאה מתקיים. ההבדל הוא שבלולאת for ידוע מספר החזרות על הלולאה בעוד שבלולאת while פרמטר זה לא ידוע.

מבנה הלולאה:

```
while(boolean condition)
{
    ......
}
```

ראשית נבדק תנאי הלולאה ובאם התנאי מתקיים מבוצע גוף הלולאה. אם לא - יוצאים מהלולאה וממשיכים באופן סדרתי בתוכנית.
שוב, אפשר לוותר על הסוגריים אם גוף הלולאה מורכב משורת קוד אחת.

נחקה את התוכנית המדפיסה את המספרים מ 1 עד 10 על ידי לולאת:while

דוגמא 9:

```
#include <stdio.h>

void main()
{
    int times = 10;
    printf("Printing the numbers form 10 to 1: \n\r");
    while (times > 0)
    {
       printf ("%d ", times);
       times--;
    }
}
```

למעשה בתוכנית זו ידענו את מספר החזרות על גוף הלולאה ולכן ניתן לתרגם פקודת while ללולאת for כך:

```
for (times = 10; times > 0; times--)
   printf ("%d ", times);
```

דוגמא 10:

```
#include <stdio.h>

void main()
{
   printf("Enter a number 1-10 (quit -0): \n\r");
   int a;
   int sum = 0;

   scanf ("%d", &a);
   while (a != 0)
   {
      sum += a;
      printf("The sum so far: %d\n\r", sum);
      scanf ("%d", &a);
   }
}
```

את לולאת ה while בתוכנית זו לא ניתן לתרגם ללולאת for מפני שמספר הפעמים שהמשתמש יקיש על מקש שונה מ 0 לא ידוע לנו ולכן במקרה זה מתאימה רק לולאת while
תנאים מורכבים בלולאה:
כמו בחלק של תנאי הלולאה בלולאת for יכולנו לרשום תנאי מורכב עם מספר משתנים כך גם בתנאי הלולאה של לולאת while עשינו זאת על ידי שימוש באופרטורים הלוגים : וגם, או כך גם כאן:
דוגמא:

```
while ( x > 47 && (y < 10 || z ==15))
```

גוף הלולאה יבוצע אם המשתנה x > 47 וגם y <10 או z == 15

לולאת do while

כמו בלולאת while פשוטה, גם כאן יש תנאי ובאם הוא מתקיים גוף הלולאה מתבצע. ההבדל הוא שראשית מבוצע גוף הלולאה ורק לאחר מכן נבדק התנאי ובאם התנאי לא מתקיים ממשיכים בתוכנית לאחר לולאת do while
כלומר תמיד גוף הלולאה יבוצע לפחות פעם אחת.

מבנה הלולאה:

```
do
{
   . . .
}
while (condition)
```

דוגמא 11:

```
#include <stdio.h>

void main()
{
    int result = 0;
    printf("The sq of the number from 1 to 10: \n\r");
    do
    {
      result += 1;
      printf("%d \n\r", result*result);
    }
    while (result < 10);
}
```

פקודת goto

תווית :(label) אוסף של פקודות המאוגדות ליחידה אחת תחת כותר מסוים, כותר זה נקרא תווית.
מבנה התווית:

```
label: command1;
    command2;
   . . .
```

פקודת goto מאפשרת קפיצה למקום אחר בתוכנית (אל תווית בתוכנית).
כללי תכנות מובנה אוסרים שימוש בפקודת goto משום שהדבר מקשה על הבנת התוכנית ועלול לגרום לטעויות

הערה:
ניתן להוכיח כי אפשר להמיר כל תוכנית המכילה goto לתוכנית ללא goto על ידי לולאת while

תבנית goto :
goto label;
כאשר label היא תווית.

דוגמא 12:

```
#include <stdio.h>

void main()
{
   int var = 0;

   a: if (var == 0)
      goto a0;
   else
      goto a1;
   a1: printf("Var != 0\n\r");
   a0: printf("Var = 0\n\r");
}
```

יש לשים לב איפה ממקדים את התווית משום שהשפה c פועלת באופן סדרתי כלומר אם במהלך התוכנית יש תווית היא תבצע אותן לפי הסדר ולפעמים זה גורם לטעויות ,דוגמא לכך:

דוגמא 12.1:

```
#include <stdio.h>

void main()
{
   int var = 0;

   a: if (var == 0)
      goto a0;
   else
      goto a1;
   a0: printf("Var = 0\n\r");
   a1: printf("Var != 0\n\r");
}
```

תיקון הטעות יכול להתבצע על ידי החלפת סדר התווית (כמו בדוגמא הקודמת) או על ידי פקודת return (תפקידה לקטוע את ריצת הפונקציה ולצאת ממנה)

נלמד לעומק על פקודה זו בפרק על פונקציות המחזירות ערך.

הפקודות break ו continue -

הפקודות break ו continue - קשורות לשלושת הלולאות שהצגנו (for, while, do while) פקודת break מפסיקה את ביצוע הלולאה אפילו אם תנאי הלולאה מתקיים. לעומת זאת פקודת continue מדלגת על הפקודות שבגוף הלולאה עד לביצוע הבא של גוף הלולאה.

תבנית לשימוש בפקודת break :

```
if (condition)
   break;
......
```

השימוש יעשה בגוף הלולאה בעזרת פקודת break ניתן לקטוע לולאות אינסופיות.
נראה זאת בדוגמא הבא:

דוגמא 13:

```
#include <stdio.h>

void main()
{
   char ch;
   while (1)
   {
      printf ("To exit print E:\n\r");
      flushall();
      scanf ("%c", &ch);
      if (ch == 'E')
      break;
   }
}
```

תבנית של הפקודה continue :

```
if (condition)
   continue;
......
```

הפקודה continue מפסיקה את המחזור הנוכחי של הלולאה ומדלגת למחזור הבא הוראות הלולאה (אם ישנן כאלו) מהפקודה continue ועד לסיומה של גוף הלולאה לא יבוצעו.

דוגמא 14:

```
#include <stdio.h>

void main()
{
    int i;
    for (i = 0; i < 10; i++)
    {
       if (i % 2 == 1)
          continue;
       printf ("%d is even\n\r", i);
    }
}
```

בס"ד

מערכים

עד כה עסקנו הצגנו בפעולות פשוטות כמו קליטת נתונים מהמשתמש. כעת נרצה לאחסן ולאגד מספר רב של נתונים בעלי טיפוס זהה - מבנה נתונים המאפשר לנו זאת הוא המערך.
מערך הינו רשימה של ערכים, כל ערך מאוחסן במקום סידורי במערך. המערך בעצם מהווה קבוצה רציפה של תאי זיכרון למשתנים מאותו סוג ועל מנת לגשת לכל תא יש לציין את מקומו הסידורי במערך.
למרות זאת כל איבר במערך מהווה משתנה בפני עצמו וכל הפעולות שנלמדו על משתנים חלים גם על איברי מערך (כמו הגדלה והקטנה עצמית וכו').

דוגמא לשימוש במערך: אם נרצה לקלוט ציונים מ- 100 תלמידים, לחשב ממוצע ציונים, לחשב את סכום הציונים, וסטיית התקן של הציונים אזי במקום להצהיר על 100 משתנים שכל אחד מהם ישמור ציון אחד, נצהיר על מערך אחת שישמור את כל 10 הציונים .
חובה לציין את גודל המערך בהצהרה (דבר המעיד על אופיו הסטטי - בפרקים מאוחרים יותר נלמד על מבנה נתונים דינאמי).

תבנית של מערך:

type array_name [length];

מערך בגודל:n
תא 0
תא 1
... ...
תאn-1
תאn

הערה:
האינדקס של המערך מתחיל מ 0 עד ל (length -1) כך שאם הצהרנו על מערך עם 10 מקומות אז האינדקס של המקום הראשון הוא 0 והאחרון 9.
כל פניה לתא מחוץ לתחום יכולה לגרום לקריסת התוכנית, שגיאה זו אינה מתגלה על ידי המהדר.

דוגמאות:
char arrayc[10]; - מערך של תווים (מחרוזת) בעל 10 מקומות
int arrayi[20]; -מערך של מספרים שלמים בעל 20 מקומות.
float arrayf[30]; - מערך של מספרים ממשיים בעל 30 מקומות.

הערה:
מערך של 10 מקומות מסוג int תופס 20 בתים בזיכרון משום ש- int תופס 2 בתים. מקומו של האיבר השלישי במערך, למשל, הוא בכתובת התחלתית של המערך + 2*3 .
מקום האיבר ה- i בזיכרון הוא: כתובת התחלתית + (גודל הטיפוס בבתים)*i.
כלומר, כאשר מוגדר מערך אין אנו יודעים את כתובתו ההתחלתית אך ידוע לנו כי כל איבריו נמצאים ברצף זה לאחר זה. לדוגמא, אם התא הראשון במערך נמצא בכתובת 2000 וזהו מערך של שלמים

בס"ד

(שגודלם 2 בתים) אזי האיבר השני ימצא בכתובת ה 2002 השלישי בכתובת 2004 וכך הלאה עד האיבר האחרון.

דרכים לאתחול מערך

ניתן לאתחל כל תא בנפרד:

```
int grades[5];
grades[0] = 90;
grades[1] = 80;
...
```

ניתן לאתחל בשורת הגדרתו על ידי רשימת ערכי המשתנים בתוך סוגריים המופרדים בניהם על ידי פסיק:

```
int grades[5] = {90, 80, ..};
```

גם כאן יש להקפיד לא לרשום מספר איברים החורג מהגודל שהגדרנו.

הערה:
ניתן לא לציין את גודל המערך אם מאתחלים בצורה זו את המערך (גודלו יחושב אוטומטית לפי מס' הערכים הנתונים).

דוגמא 1:

```
#include <stdio.h>
#define LENG 5

void main()
{
    int grades [LENG];
    int i, passed = 0;
    int sum = 0;
    float avg = 0;

   for (i = 0; i < LENG; i++)
   {
      printf("Enter your grade: \n\r");
      scanf("%d", &grades[i]);
      if (grades[i] > 60)
         passed++;
         sum += grades[i];
    }
    avg = sum/LENG;
    printf("The average is: %f", avg);
    printf("\n\r%d student passed the test",passed);
}
```

בס"ד

הערה:
ניתן לאתחל את כל ערכי משתני המערך ב- 0 באופן הבא:
type arr [length] = {0}; אך בעבור כל ערך אחר הדבר בלתי אפשרי באופן הנ"ל אלא יש צורך בלולאה שתעבור ותאתחל את כל עברי המערך בערך הדרוש.

מערך מחרוזת

מחרוזת היא מערך של משתנים מסוג char כאשר כל מחרוזת ב c -מסתיימת בקוד '\0' - זהו null terminator המציין כי כאן מסתיימת המחרוזת ולכן המקום שיוקצה למחרוזת בעלת n תווים היא n+1 .

דוגמאות לאתחול מערך ולהצגתו:

דוגמא 2:

```
#include <stdio.h>

void main()
{
    char name[6];

    name[0] = 'H';
    name[1] = 'e';
    name[2] = 'l';
    name[3] = 'l';
    name[4] = 'o';
    name[5] = '\0';

    puts(name);
    printf ("%s\n\r", name);

    char name1[] = {'W', 'o', 'r', 'l', 'd', '\0'};
    putchar(name1[0]);
    putchar(name1[1]);
    putchar(name2[2]);
    putchar(name1[3]);
    putchar(name1[4]);

    putchar('\n');


    char name2[] ={"Goodbye"};

    puts(name2);
}
```

הערה '\n': הינו תו מעבר לשורה חדשה.

- puts(string) זוהי פונקצית פלט של מחרוזת שמקבלת כארגומנט מחרוזת, מדפיסה אותה ועוברת לשורה חדשה. זהה בביצוע לפונקצית הפלט printf כאשר בסיומה אנו מוסיפים (\n\r הפונקציה מדפיסה תו-תו עד אשר היא פוגשת '\0'

- gets(string) זוהי פונקצית קלט של מחרוזת שמקבלת משתנה מחרוזתי ומעדכנת את ערכו (קולטת עד אשר היא פוגשת '\0').

דוגמא 3:

```
#include <stdio.h>

void main()
{
    char name[10];
    char l_name[10];
    printf ("Enter your name: ");
    gets (name);
    printf ("Enter your last name: ");
    gets (l_name);
    printf("Your full name is: ");
    puts (name);
    puts (l_name)
}
```

הערה:
בשימוש ב gets(string) -יש לשים לב לא לקלוט שם החורג מגבולות המערך שכן כבר הזכרנו שהדבר עלול לגרום לשגיאות חמורות. כמו כן לפני שימוש חוזר בפונקציה זו יש לנקות את החוצץ ע"י קריאה לפונקציה flushall() כאשר יש שאריות בחוצץ

הערה חשובה:
אין השמה של מחרוזת מכיוון שמחרוזת היא מערך של תווים.

יש צורך בהשמה של כל איבר במערך.

הספרייה string.h

ספרייה לטיפול במחרוזות, נראה מספר פונקציות בספריה זו ונדגים את פעולתן.

- strlen (string) פונקציה המחזירה את מספר התווים במחרוזת string לא כולל התו "\0"

- strcpy (string1, string2)

בס"ד

הפונקציה מקבלת 2 מחרוזת ומעתיקה את תוכנה של המחרוזת השנייה לראשונה (כלומר תוכנה של המחרוזת string1 אחרי פקודה זו יהיה התוכן של string2

לדוגמא:

```
char first[ ] = "ab";
char second[ ] = "cd";
strcpy(first, second);
```

– strcat (string1, string2)

הפונקציה מקבלת 2 מחרוזות ומשרשרת את תוכנה של המחרוזת השנייה לראשונה

first יכיל את המחרוזת "cd"

תוכנה של המחרוזת string1 אחרי פקודה זו יהיה תוכנה המקורי והמשכה יהיה המחרוזת string2
לדוגמא:

```
char first[5] = "ab"
char second[ ] = "cd"
strcat (first, second)
```

כעת first יכיל "abcd"

– strcmp (string1, string1)

הפונקציה מקבלת 2 מחרוזות ומחזירה 0 אם המחרוזות זהות, ערך חיובי אם string1 > string2
וערך שלילי אם string1 < string2
לדוגמא strcmp("abc", "abc"): מחזירה 0.

- atoi (string)

הפונקציה מקבלת מחרוזת string ומחזירה את ערכה המספרי.
לדוגמא: atoi("12") תחזיר את המספר השלם 12

– itoa (num, string, baes)

הפונקציה מקבלת מספר, מחרוזת ומספר המייצג בסיס ספירה ומחזירה מחרוזת מספרית שהיא ההצגה בבסיס הספירה של המספר הנתון.
לדוגמא: itoa (15, str, 2) תחזיר את המחרוזת "111"

הערה:
שתי הפונקציות האחרונות הן מתוך הספרייה stdlib.h

דוגמא 4:

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

void main()
{
      char name[20];
      char l_name[10];
      printf ("Enter your name: ");
      gets (name);
      printf ("Enter your last name: ");
      gets (l_name);
      printf("Your full name is: ");
      strcat (name, " ");
      strcat (name, l_name);
      puts(name);
      printf ("The length of your name is: %d\n\r", strlen(name));
      if (!strcmp (name, "will smith"))
         puts("Hi");
      else
         puts("Bye");
}
```

דוגמא 5:

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


void main()
{
   char name[20];
   int leng, i;

   printf ("Enter your name: ");
   gets (name);
   leng = strlen(name);

   printf ("Your reverse name is:");
   for (i = leng; i >= 0; i--)
   putchar(name[i]);
}
```

ניתן גם להשתמש במערכים דו מימדים - לדוגמא int matrix[5][5] הינו מערך דו מימדי בעל 25 תאים מסוג שלם המכיל 5 שורות ו- 5 עמודות.
כל הכללים למערך חד מימדי תקפים גם בעבור מערך דו מימדי.

דוגמא 6:

```
#include <stdio.h>

void main()
{
   int matrix[10][10];

   int i,j;

   for (i = 1; i <= 10; i++)
      for (j = 1; j <= 10; j++)
         matrix[i-1][j-1] = i*j;


   for (i = 0; i < 10; i++)
   {
      putchar('\n');
         for (j = 0; j < 10; j++)
            printf ("%2d ", matrix [i][j]);
   }

}
```

בס"ד

דוגמא 7:

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

void main()
{
    int arr[5][5];
    int temp[5][5];

    int i,j;

   for (i = 0; i < 5; i++)
     for (j = 0; j < 5; j++)
         arr[i][j] = random() % 100 +1; //creating a random array

    printf ("Printing the array: \n");
    for (i = 0; i < 5; i++)
    {
       putchar('\n');
       for (j = 0; j < 5; j++)
          printf ("%2d ", arr[i][j]);
    }

    printf ("\n Printing the array backword: \n");
    for (i = 4; i >= 0; i--)
    {
       putchar('\n');
       for (j = 4; j >= 0; j--)
          printf ("%2d ", arr[i][j]);
    }

    printf ("\nChanging the array by using a temporary array:\n");
    for (i = 0; i < 5; i++)
       for (j = 0; j < 5; j++)
          temp[i][j] = arr[i][j];

    for (i = 0; i < 5; i++)
       for (j = 0; j < 5; j++)
          arr[i][j] = temp[4-i][4-j];


    printf ("\nThe new array is: \n");
    for (i = 0; i < 5; i++)
    {
       putchar('\n');
       for (j = 0; j < 5; j++)
```

```
        printf ("%2d ", arr[i][j]);
    }
}
```

דוגמא 8:

```
#include <stdlib.h> //for atoi
#include <stdio.h>

int main() {

    char buf1[] = "597332";
     int x = atoi(buf1);
     printf ("decimal: %d\n",x);
     char buf2[16] = {0};
     int ret = sprintf(buf2, "%d", x);
     printf ("decimal: %s\n",buf2);

    return 0;
}


/* itoa example */
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  int i;
  char buffer [33];
  printf ("Enter a number: ");
  scanf ("%d",&i);
  itoa (i,buffer,10);
  printf ("decimal: %s\n",buffer);
  itoa (i,buffer,16);
  printf ("hexadecimal: %s\n",buffer);
  itoa (i,buffer,2);
  printf ("binary: %s\n",buffer);
  return 0;
}
```

פונקציות

בפרק הראשון טענו כי שפת c הינה שפה פרוצדוראלית, משמע שניתן לפרק את התוכנית לפונקציות - כל פונקציה בעלת תפקיד מוגדר וניתן להשתמש בה על ידי קריאה לפונקציה.
היתרון הוא פשטות הקריאה של התוכנית והבנתה.
פונקציה אם כך היא קטע קוד בעל שם ייחודי (כמו main) בעלת מטרה ברורה, לדוגמא - פונקציה הקוראת נתונים מהמקלדת, מדפיסה, ממיינת וכו'.

תבנית ההגדרה של פונקציה:
returned_type function_name (list_of_parameters)

returned_type - טיפוס הערך שהפונקציה מחזירה. לדוגמא, הפונקציה main אינה מחזירה ערך ולכן טיפוס הערך המוחזר שלה יהיה void (ריק).
function_name - שם הפונקציה. יש לתת שם משמעותי שיעיד על תפקיד הפונקציה. כפי שציינו שם הפונקציה הראשית הוא main ואינו ניתן לשינוי.
list_of_parameters - רשימת פרמטרים (משתנים), מופרדים על ידי פסיקים. הפרמטרים הם ערכי המשתנים להם זקוקה הפונקציה על מנת לבצע את תפקידה.לדוגמא, אם נבנה פונקציה המחשבת סכום האיברים במערך אזי נשלח את המערך כפרמטר לפונקציה, הפונקציה תכיר במערך ותוכל לחשב את סכומו.
במקרה ואין צורך בפרמטרים נשאיר את הסוגריים של הפונקציה ריקים.

הגדרה:
ארגומנטים - המשתנים המוצבים בעת הקריאה לפונקציה.
פרמטרים - הם המשתנים שמופיעים בתוך הסוגריים של הפונקציה. לתוכם מוצבים הארגומנטים שהתוכנית שולחת.

כאשר אנו שולחים כפרמטר מערך דו מימדי ניתן להשמיט את המימד הראשון אך השני חייב להופיע.
בהעברת מערך חד מימדי כפרמטר ניתן להשמיט את גודלו.

כאשר כותבים את הפונקציות מעל לפונקציה הראשית (main) אין צורך בהצהרה מוקדמת על פונקציה. לעומת זאת אם הפונקציה נכתבת לאחר הפונקציה הראשית יש צורך להצהיר עליה (כדי שהפונקציה הראשית תכיר אותה) כפי שהגדרנו תבנית הגדרה של פונקציה או ללא שם הפרמטר.

לדוגמא:

```
int sum (int s);
```
או גם כך:
```
int sum (int);
```

בשני המקרים המהדר יידע שהפונקציה תוגדר בהמשך.

בכתיבת הפונקציה ובהגדרתה ניתן להשתמש בשמות שונים לארגומנטים כלומר, אם נשלח לפונקציה a משתנה ששמו b לא חייב השם b להופיע אח"כ בפונקציה אלא ניתן להשתמש בשמות אחרים. זאת מכיוון שערכו של המשתנה הוא זה שמועתק לפרמטר של הפונקציה, כלומר עם ביצוע הקריאה לפונקציה מועתקים זה אחר זה ערכי הארגומנטים המצויים בפקודה הקוראת אל הפרמטרים של הפונקציה הנקראת ומאתחלים אותם.
כאשר קוראים לפונקציה אנו חייבים לספק לה את הערכים הדרושים בהתאם לפרמטרים השונים של

הפונקציה כך שבזמן שאנו קוראים לפונקציה ערכים אלו מועתקים לפרמטרים השונים וניתן לשנות את הערכים של הפרמטרים מבלי שהדבר יפגע בערכים ששלחנו.
ניתן לקרוא לפונקציה מספר לא מוגבל של פעמים.

כאשר מעבירים משתנים לפונקציה כפרמטרים, ערכי המשתנים לא משתנה לאחר ביצוע הפונקציה.
עם זאת, מאחר ובמערך המצב שונה (לא מתבצעת העתקה, כפי שכבר הוזכר), גם בקריאה לפונקציה לא נעשית העתקה של ארגומנט מערך לפרמטר של הפונקציה ושינוי שמבצעים בפונקציה בערכי המערך ישאיר את ערכי המערך עם השינוי גם לאחר סיום ביצוע הפונקציה.

דוגמא 1:

משולש פסקל - כידוע מכיל את מקדמי הבינום של ניוטון, כל איבר מתקבל מסכום שני האיברים שמתחתיו. יצרנו מערך בגודל 9X9 אבל השתמשנו בחציו, כל שאר התאים במערך מכילים 0.

למשל:

$n=0$					1			
$n=1$				1		1		
$n=2$			1		2		1	
$n=3$		1		3		3		1
$n=4$	1		4		6		4		1
$n=5$		1		5		10		10		5		1
$n=6$	1		6		15		20		15		6		1

```
#include <stdio.h>

const int LENG = 9;

void creating_arr (int paskal[][LENG])
{
   int i,j;
   for (i = 0; i < LENG; i++)
      for (j = 0; j < i+1; j++)
      {
         if (j == 0 || j == i)
            paskal[i][j] = 1;
         else
            paskal[i][j] = paskal[i-1][j-1]+paskal[i-1][j];
      }
}

void writing_arr (int paskal[][LENG])
{
   int i,j;
   for (i = 0; i < LENG; i++)
   {
      printf("\n");
      for (j = 0; j < i+1; j++)
         printf("%d ", paskal[i][j]);
   }
```

```
}

void main()
{
    int paskal[LENG][LENG] = {0};
    creating_arr (paskal);
    writing_arr (paskal);
}
```

הערה:
הצהרנו על "משתנה" מסוג קבוע (const) משמע ערכו לא יוכל להשתנות במהלך התוכנית (מקובל לכתוב קבוע באותיות גדולות).
ניתן להצהיר על קבוע מכל טיפוס שהוא : שלם, ממשי, תווי וכו.'
כאשר מעוניינים שערכם של הארגומנטים ישתנו אז יש לכתוב את אופרטור הכתובת (&) לפני הארגומנט ששולחים וכן בהצהרה על הפונקציה יש לכתוב כפרמטר את סוג הטיפוס ואחריו * (הסבר מלא יובא כשנדבר על מצביעים - כרגע חשוב להבין שכאשר אנו משנים את ערכו של משתנה בתוך פונקציה ומעוניינים ששינוי זה ישמר גם מחוץ לפונקציה יש לשלוח כארגומנט את המשתנה ולפניו לרשום & ובפונקציה להצהיר על הפרמטר ששלחנו (עם &) עם כוכבית לדוגמא שלחנו &x ו x משתנה שלם אזי בהצהרה של הפונקציה הפרמטר יהיה int* x

משפט return

זהו הערך המוחזר על ידי הפונקציה. כאשר הפונקציה אינה מטיפוס void אז היא צריכה להכיל לפחות משפט return אחד שיופיע כך return exp :ערכו של הביטוי exp מחושב וידוע. ביצוע של משפט זה מסיים את פעולתה של הפונקציה וחוזרים להמשך הפונקציה שקראה לה.

הערה:
פונקציה מטיפוס void נקראת פרוצדורה.
ניתן לרשום בפרוצדורה משפט return על מנת לסיים אותה אך ללא ערך מוחזר.

בס"ד

דוגמא 2:

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

const int LENG = 9;

void creating_arr (int matrix[][LENG])
{
    int i,j;
    for (i = 0; i < LENG; i++)
       for (j = 0; j < LENG; j++)
          matrix[i][j] = random(40);
}

void printing_arr (int matrix[][LENG])
{
    int i,j;
    printf("The array is: \n");
    for (i = 0; i < LENG; i++)
    {
       printf("\n");
       for (j = 0; j < LENG; j++)
          printf("%3d ", matrix[i][j]);
    }
}

int sum_arr (int matrix[][LENG])
{

    int i, j, sum = 0;


    for (i = 0; i < LENG; i++)
       for (j = 0; j < LENG; j++)
          sum += matrix[i][j];
    printf ("\n\rSum of array is:");
    return sum;
}

int max_arr (int matrix[][LENG])
{

    int i, j, max = matrix[0][0];


    for (i = 0; i < LENG; i++)
```

```
      for (j = 0; j < LENG; j++)
         if (max < matrix[i][j])
            max = matrix[i][j];
   printf ("\n\rMax of array is:");
   return max;
}

void main()
{
   randomize();
   int arr[LENG][LENG];
   creating_arr (arr);
   printing_arr (arr);

   printf(" %d ",sum_arr (arr));
   printf ("%d", max_arr (arr));

}
```

בס"ד

הערה:
גם פונקציית main יכולה להיות מטיפוס שונה מ void ולהחזיר ערך.
לדוגמא:

```
int main()
{
    . . .
    return 0;
}
```

דוגמא 3:

```
#include <stdio.h>

long factorial (int n);
int chack_num (int num);

void main()
{
    int num;
    long fuc;

    printf("Enter a number 0-9: ");
    scanf ("%d", &num);

    while (!chack_num (num))
    {
       printf ("\n\rWrong number Try again: ");
       scanf ("%d", &num);
    }

    fuc = factorial(num);

    printf ("\n\rThe factorial of your number is: %ld",fuc);
}

int chack_num (int num)
{
    if (num > 9 || num < 0)
       return 0;
    else
       return 1;
}
```

```
long factorial (int n)
{
   int i = 1;
   double fuc = 1;
   for ( ; i <= n ; i++)
      fuc *= i;
   return fuc;
}
```

טווח ההכרה

כפי שציינו בפרקים קודמים פונקציה (או משתנה) מוכר ממקום הגדרתו ועד סוף התוכנית כך שכל פונקציה יכולה לקרוא לפונקציות מוכרות לה (אלה שכתובות מעליה ולכן נהוג לכתוב את הפונקציה הראשית אחרונה).

דוגמא:

```
void a()
{
   . . .
}
void b()
{
   . . .
}
void c()
{
   . . .
}
void main()
{
   . . .
}
```

פונקציה b מכירה רק את פונקציה a
פונקציה c מכירה את הפונקציות a, b
פונקצית main מכירה את הפונקציות a, b, c

דיברנו בפרקים קודמים על משתנים גלובאליים ולוקאליים. נזכיר כי משתנים גלובליים מוכרים בכל התוכנית ממקום הגדרתם והלאה ומשתנים לוקאליים מוכרים בתוך הפונקציה שלהם בלבד - אך יש לציין כי מקומם בזיכרון משתחרר עם סיום הפונקציה.

בס"ד

דוגמא 4:

```
#include <stdio.h>

int x = 3;

void a (int x)
{
    x = 5;
    printf ("\n\rThe value of x is: %d", x);
}


void b (int* x)
{
    (*x)++;
    printf ("\n\rThe value of *x is: %d", *x);
}

void c()
{
    x= x+2;
    printf ("\n\rThe value of x is: %d", x);
}

void main()
{
    printf ("\n\rThe value of x is: %d", x);
    a (x);
    printf ("\n\rThe value of x after function a is: %d", x);
    b (&x);
    printf ("\n\rThe value of x after function b is: %d", x);
    {

      int x = 1;

      c();
      printf ("\n\rThe value of x after function c is: %d", x);
   }
}
```

דוגמא 5:

```
// C program to sort the array in an
// ascending order using selection sort

#include <stdio.h>

void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// Function to perform Selection Sort
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n - 1; i++) {

        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element
        // with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

// Function to print an array
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver code
int main()
{
    int arr[] = { 0, 23, 14, 12, 9,2 };
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: \n");
```

```
    printArray(arr, n);

    selectionSort(arr, n);
    printf("\nSorted array in Ascending order: \n");
    printArray(arr, n);

    return 0;
}
```

בס"ד

קבצים

בתוכניות מחשב רבות יש צורך לשמור את הנתונים - תוכנות כמו מעבדי תמלילים, יומני פגישות, הנהלת חשבונות וכדומה שומרות נתונים לשימוש חוזר. בכל התוכניות שהצגנו שמרנו את הנתונים בזיכרון של המחשב, אך כאשר כיבינו את המחשב הנתונים הללו נמחקו מהזיכרון. כאשר אנו כותבים מסמך במעבד תמלילים נרצה לשמור את המידע בצורה כלשהוא ובמידת הצורך לטעון את המידע בחזרה לזיכרון גם לאחר זמן רב.
קיימים התקני אחסון חיצוניים לזיכרון המחשב: דיסק קשיח, תקליטון או תקליטור אופטי ועוד.
באמצעים אלה ניתן לשמור את הנתונים ללא תלות בכיבוי המחשב או בסיום בלתי צפוי של התוכנית.
לשם כך נרכז את הנתונים שנרצה לשמור **בקבצים**.
קובץ - אוסף של בתים שיש קשר איכות בניהם ויש לו שם שנקבע על ידי המשתמש. מערכת ההפעלה היא האחראית לנהל את השמירה של הנתונים בקבצים ואת הטעינה שלהם (היא עושה זאת על ידי ניהול טבלה בה רשומים שמות הקבצים ומיקומם).

קיימים שני סוגי קבצים: קבצי טקסט וקבצים בינאריים. בפרק זה נעסוק בקבצי טקסט, נראה כיצד פותחים קובץ, כותבים בו או קוראים ממנו וסגירתו.
ברב הקבצים קיים פורמט מסוים שמוכר לתוכניות שכותבות או קוראות ממנו. פורמט זה מציין כיצד המידע מאוחסן בקובץ. בכל תוכנית ניתן לקבוע פורמט עבור סוג הנתונים שצריך לשמור בו - לדוגמא, אם נכתוב תוכנית לניהול ספר טלפוניים נוכל לבנות את הפורמט כך ששדהו הראשון יהיה מספר הרשומות בקובץ, שדהו השני מספר המציין את סוג המיון (מספרי ,אלפבתי), שדהו השלישי תו המציין אם הנתונים הם בעברית או באנגלית ועוד שדות בהתאם לרצוננו.

קובץ טקסט

סוג קובץ זה מיועד להיקרא על ידי כל תוכנית המסוגלת לקרוא קבצי טקסט כמו edit, dos,מעבדי תמלילים ועוד. בקובץ זה לא מופיעים תווים מיוחדים כמו null.
הוא ערוך בשורות אשר מופרדות על ידי התווים '\n' ו '\r' ובסיום כל קובץ נמצא גם התו eof המציין סוף קובץ (end of file).
כדי לגשת לקובץ נגדיר משתנה FILE* f שהוא מצביע לקובץ (מצביע *). תו הכוכבית מציין כי f אינו קובץ בעצמו אלא מצביע לראשיתו (בפרקים על מבני נתונים נעסוק בהרחבה במצביעים, כרגע חשוב רק להבין שכדי לקרוא\לכתוב לקובץ נציין את ראשיתו על ידי אופרטור הכוכבית).

מבנהו של file בספרייה stdio:

```
typedef struct
{
    short evel;
    unsigned flags;
    char fd;
    unsigned char hold;
    short bsize;
    unsigned char *buffer, *curp;
    unsigned istemp;
    short token;
}FILE;
```

נסביר את חלקם:
- fd מספר הזיהוי של הקובץ במערכת ההפעלה.
- buffer החוצץ שדרכו מתבצעת הקריאה והכתיבה מהקובץ.
- curp מצביע על המיקום הנוכחי בקובץ.

סוגי גישה לקובץ

- r קריאה בלבד מקובץ קיים.
- w כתיבה בלבד, אם הקובץ קיים הוא ימחק.
- a הוספה לקובץ קיים. אם לא קיים נוצר חדש כך שהפעולה של פקודת append שקולה ליצירת קובץ חדש.
- r+ קריאה וכתיבה לקובץ קיים.
- w+ריאה וכתיבה לקובץ אם הקובץ קיים הוא ימחק.
- a+הוספה לקובץ ואפשרות קריאה ממנו.

פתיחה וסגירה של קבצים

הפונקציה :()fopen
כדי לקרוא או לכתוב לקובץ יש צורך בפתיחתו. פתיחת הקובץ נעשית על ידי קריאה לפונקציה) fopenפונקציה זו מצויה בספריה (stdio המקבלת כפרמטר שם קובץ וסוג גישה (קריאה\ כתיבה).
לאחר תו זה נוסיף תו שיציין אם הקובץ הוא קובץ טקסט או בינארי, אם הקובץ הוא קובץ טקסט נוסיף את התו t, אם הקובץ הוא קובץ בינארי נוסיף את התו.b
לדוגמא: קיים קובץ טקסט הנקרא , file"" נפתח אותו לקריאה כדלהלן:

```
FILE *f = fopen ("file.txt", rt");
```

על פי התבנית לפתיחת קובץ.

התבנית לפתיחת קובץ:

```
FILE *f = fopen (file_name, access_type);
```
או:
```
FILE *f;
f = fopen (file_name , access_type);
```

(נדון בפירוט בחלק של מבנה הנתונים במצביעים)FILE הוא משתנה מסוג מצביע למבנה f כאשר שם קובץ חוקי מורכב משם וסיומת המעידה על אופיו כמו -: bat, tif, gif, jpeg וכו' txt המופרדים בנקודה. בקבצי טקסט הסיומת המקובלת היא .

חשוב מאוד לבדוק אם פתיחת הקובץ הסתיימה בהצלחה או נכשלה.
כאשר פתיחת הקובץ נכשלה ולכן נבדוק שהערך המוחזר null מחזירה fopen הפונקציה
כישלון בפתיחת הקובץ יכול להתרחש מכמה סיבות. null- מהפונקציה שונה מ:
1. שם קובץ לא חוקי
2. לא קיים קובץ עם שם זה
3. ניסיון לפתוח קובץ לכתיבה כאשר הוא קובץ לקריאה בלבד
4. אין הרשאות לפתיחת הקובץ

כאשר פתיחת הקובץ חיונית לתוכנית ופעולה זו נכשלה נרצה לסיים את התוכנית לשם כך נשתמש בפונקציה exit המופיעה בספרייה stdlib כך:

```
FILE *tf = fopen ("file_name.txt", "wt");
if (tf == null)
{
    printf("failed to open the text file");
    exit (1);
}
```

הפונקציה fclose(): לאחר השימוש בקובץ יש לסגור אותו. הפונקציה שמבצעת זאת היא fclose המקבלת כארגומנט מצביע לקובץ. הפונקציה 0 אם הסגירה הצליחה stdio הנמצאת בספרייה המציין -1 אחרת EOF (end of file).

התבנית לסגירת קובץ:
fclose (poiner_2_file);

הפונקציה feof (pointer_to_file):
stdio הנמצאת גם היא בספרייה feof על מנת לבדוק אם הגענו לסוף הקובץ יש להשתמש בפונקציה ומקבלת כארגומט מצביע לקובץ, מחזירה 0 אם לא הגענו לסוף הקובץ אחרת ערך שונה מ- 0.

כתיבה וקריאה מקבצים

fprintf (pointer_to_file, " …", list_of_variables)
f פונקצית כתיבה לקובץ, דומה לפונקצית כתיבה למסך (התקן הפלט הסטנדרטי) רק בתוספת בתחילה המציינת כי יש לכתוב לקובץ (file).
printf. היא מקבלת כארגומנט מצביע לקובץ, ואח"כ הכל זהה לפונקציה

fscanf (pointer_to_file, "..", list_of_variables)
כמו כן הפונקציה לקריאה מהקובץ דומה לפונקצית קריאה מהמקלדת (התקן הקלט הסטנדרטי) בתוספת f. כמו ב fprintf -מהסיבה שהוזכרה. יש לשלוח כארגומנט את המצביע לקובץ.

גם כאשר רוצים לקלוט נתונים מהתקן הקלט הסטנדרטי (מקלדת) fprintf ניתן להשתמש בפונקציה כאשר בשדה fscanf כנ"ל לגבי stdin. נכתוב fprintf על ידי כך שבשדה הראשון של הפונקציה הראשון נכתוב stdout.
דוגמא 1:

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

FILE *f;
char file_name[15];
char c;

void write_2_file (char file_name[])
{
    f = fopen(file_name, "wt");
```

```
    if (f == NULL)
      exit(1);

    else
    {
      printf("The file %s has been opened \n\r", file_name);
      printf ("Enter text to the file \n\r");
      fflush (stdin); //cleaning the buffer

      for (fscanf(stdin, "%c", &c); !feof (stdin); fscanf (stdin, "%c", &c))
        fprintf (f, "%c", c);
      if (fclose(f))
        exit (1); //the closing wasnt sucssesful
      else
        printf ("\n\rThe file %s has been closed\n\r", file_name);

    }
}

void read_from_file (char file_name[])
{
    FILE *f;
    printf("Reading from the file %s :\n\r",file_name);
    f = fopen (file_name, "rt");

    if (f == NULL)
    {
      printf ("Can't open the file %s ", file_name);
      exit(1);
    }

    else

    {
      printf("The file %s has been opened \n\r", file_name);
      for (fscanf(f,"%c", &c); !feof (f); fscanf (f,"%c", &c))
        fprintf (stdout, "%c", c);
      if (fclose(f))
        exit (1); //the closing wasnt sucssesful
      else
        printf ("\n\rThe file %s has been closed", file_name);
    }


}

void main()
{
    printf ("\n\rEnter file name to create: ");
```

```
    scanf ("%s", file_name);

    write_2_file (file_name);
    read_from_file (file_name);
}
```

דוגמא 2:

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

FILE *from_f;
FILE *to_f;
char from_file[15];
char to_file[15];

char c;

void coping (char from_file[], char to_file[])
{
    from_f = fopen (from_file, "rt");
    to_f = fopen (to_file, "wt");

    if ((from_f == NULL) || (to_f == NULL))
      exit(1);

    else
    {
      printf("The files %s and %s has been opened \n\r", from_file, to_file);
      printf("The text in the file %s\n\r", from_file);

      for (fscanf(from_f, "%c", &c); !feof (from_f); fscanf (from_f, "%c", &c))
      {
        fprintf (to_f, "%c", c);
        fprintf (stdout,"%c", c);
      }

      if ((fclose(to_f)) || (fclose (from_f)))
        exit (1); //the closing wasnt sucssesful
      else
        printf ("\n\rThe files %s and %s has been closed\n\r", from_file, to_file);
    }
}

void main()
```

```
{
    printf ("\n\rEnter file name to copy: ");
    scanf ("%s", from_file);
    printf ("\n\rEnter a new file name: ");
    scanf ("%s", to_file);
    coping (from_file, to_file);
}
```

דוגמא 3:

התוכנית קולטת שמות של שני קבצים ובודקת האם הם זהים בתוכנם.

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

FILE *f1;
FILE *f2;
char first_f[15];
char second_f[15];

char c1,c2;
int flag = 1;

void compare (char first_f[], char second_f[])
{
    f1 = fopen (first_f, "rt");
    f2 = fopen (second_f, "rt");

    if ((f1 == NULL) || (f2 == NULL))
       exit(1);
    else
    {
       printf("The files %s and %s has been opened \n\r", first_f, second_f);

       for ( (fscanf(f1, "%c", &c1)) && (fscanf (f2, "%c", &c2));
          (!feof (f1)) && (!feof (f2));
          (fscanf (f1, "%c", &c1))&& (fscanf (f2, "%c", &c2)) )
          {
             if (! c1 == c2)
             {
                printf ("\n\rThe two files are not the same ");
                flag = 0;
                break;
             }
             fprintf (stdout,"%c", c1);
          }
```

```
        if (flag)
          printf("\n\rThe two files are the same");
        if ((fclose(f1)) || (fclose (f2)))
          exit (1); //the closing wasnt sucssesful
        else
          printf ("\n\rThe files %s and %s has been closed\n\r", first_f, second_f);
    }
}

void main()
{
    printf ("\n\rEnter the first file name: ");
    scanf ("%s", first_f);
    printf ("\n\rEnter the second file name: ");
    scanf ("%s", second_f);
    compare (first_f, second_f);
}
```

פונקציות נוספות לשימוש בקבצים

fgetc(pointer_to_file) - קריאת תו מקובץ.

fputc(char, pointer_to_file) - כתיבת תו לקובץ.

fgets(string, max_leng, pointer_to_file) - קריאת מחרוזת לתוך משתנה מחרוזת (string) מקובץ.

fputs(string, pointer_to_file) - כתיבת שורה שלמה לתוך קובץ.

השימוש בהם נעשה בדומה לשימושים של הפונקציות שהצגנו בפרק הקלט/פלט (putc, puts, gets, getc וכו').

דוגמא 4:

תוכנית הבודקת את מס' המופעים של תו נתון בקובץ נתון.

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

FILE *f;

void count (char ch, char file_n[])
{
    int num = 0;
    char c;
```

```
    if ((f = fopen (file_n, "rt")) == NULL)
    {
      printf("\n\rCan't open file %s ", file_n);
      exit(1);
    }
    else
    {
      printf("The file %s has been opened \n\r", file_n);

      for (c = fgetc(f); !feof (f); c = fgetc (f))
      {
        if (c == ch)
          num++;
        else
          continue;
      }
      printf("\n\rThe are %d %c in the file %s",num ,ch, file_n);
      if (fclose(f))
        exit (1); //the closing wasnt sucssesful
      else
      printf ("\n\rThe files %s has been closed\n\r", file_n);
    }
}

void main()
{
    char ch;
    char file_name[15];

    printf ("\n\rEnter the char : ");
    scanf ("%c", &ch);
    printf ("\n\rEnter the name of a file: ");
    scanf ("%s", file_name);
    count (ch, file_name);
}
```

בס"ד

מבנים

בפרק זה נדון במבנה נתונים הנקרא מבנה. מה שמאפיין אותו הוא שניתן במשתנה אחד להכיל מספר רב של נתונים מטיפוסים שונים (כל משתנה בתוך המבנה נקרא שדה). ניתן לומר כי מבנה דומה למערך בכך ששניהם יכולים לאגד מספר נתונים, אך מבנה נותן לנו גם חופש בבחירת טיפוסי הנתונים ,כלומר מבנה יכול להכיל גם float, int, doudle, file ועוד בעת ובעונה אחת תחת משתנה אחד מטיפוס המבנה.

התבנית של מבנה:

```
struct name_of_struct
{
    variables
};
```

בהגדרת המערך יש לכתוב את המילה השמורה struct ולאחר מכן את שם המבנה שבחרנו (חשוב כי שמו של המבנה יהיה משמעותי בדיוק כמו בבחירת שמות למשתנים). בתוך הסוגריים יש לרשום את המשתנים ובסוף נקודה-פסיק.(;)
לדוגמא:

```
struct address
{
    char city[20];
    char street[20];
    int num_house;
    long code_area;
};
```

יצרנו מבנה של כתובת הכולל: שם עיר ורחוב- מחרוזות בעלות 20 תווים, מספר בית- משתנה מסוג שלם, ומיקוד- מספר שלם.
כעת נצהיר על משתנה מהטיפוס כתובת:
struct address add;
כמו כן כשנרצה לגשת לשדות של המשתנה add ולעדכן אותם, נוכל לעשות זאת על ידי האופרטור נקודה (.) לדוגמא נציב את בשדה מס' הבית 5 כך:
add.num_house = 5;
כך ניתן לעדכן ולשנות את השדות.

ניתן להצהיר על מספר משתנים מטיפוס מבנה בשורה אחת כמו שהצהרנו על משתנים פשוטים לדוגמא.struct address add1, add2, add3 :
כמו כן ניתן לאתחל מבנים כפי שאתחלנו מערך:

struct type name = {value1, value2,. . .};

לדוגמא:
struct address add = {"migdal Haemek", "Harazim", 5, 23028};

נסכם ונאמר כי ניתן לבצע את כל הפעולות שביצענו במשתנים פשוטים על מבנה באותו אופן (הצבה כארגומנט בפונקציה, הצהרה וכו').

מבנה כפרמטר של פונקציה

נראה שני סוגים של מבנה כפרמטר של פונקציה האחד העברה לפי תוכן והשנייה לפי כתובת:

העברה לפי תוכן:

```
void print_detailes (struct person p1, struct person p2,struct person p3)
{
    printf ("\n\rThe detailes of the tree persons: \n\r");
    printf ("%s %s %d \n\r", p1.f_name, p1.l_name, p1.age);
    printf ("%s %s %d \n\r", p2.f_name, p2.l_name, p2.age);
    printf ("%s %s %d \n\r", p3.f_name, p3.l_name, p3.age);
}
```

נניח כי אנו שולחים לפונקציה הזו 3 משתנים מסוג מבנה (המבנה הוא של אדם ושדותיו: שם פרטי, שם משפחה וגיל) אזי כדי להדפיס את שדותיהם יש פשוט להשתמש באופרטור הנקודה ובכך לפנות לשדות של המשתנים.

העברה לפי כתובת:

```
void get_detailes (struct person* p1, struct person* p2, struct person* p3)
{
    printf ("Enter first name: ");
    gets ((*p1).f_name);
    printf ("Enter last name: ");
    gets ((*p1).l_name);
    printf ("Enter age: ");
    scanf ("%d", &(*p1).age);

    printf ("Enter first name: ");
    gets ((*p2).f_name);
    printf ("Enter last name: ");
    gets ((*p2).l_name);
    printf ("Enter age: ");
    scanf ("%d", &(*p2).age);

    printf ("Enter first name: ");
    gets ((*p3).f_name);
    printf ("Enter last name: ");
    gets ((*p3).l_name);
    printf ("Enter age: ");
    scanf ("%d", &(*p3).age);
}
```

נניח כי בפונקצית ()main יש גם 3 משתנים מסוג person : pr1,pr2,pr3 וכתובותיהם בהתאמה FFD6, FFB6, FF96. כשנעביר את המשתנים לפונקציה get_detailes נרצה שתוכנם של המשתנים pr1,pr2,pr3 ישתנו ולכן נעביר את הכתובת של המשתנים (pr1,pr2,pr3 כי משתנה מטיפוס מבנה מתנהג כמו משתנה מטיפוס פשוט int, float :וכו' ולכן אם נעביר את תוכנו של המשתנה מטיפוס מבנה הארגומנט שנשלח לא ישתנה בעקבות הפונקציה שאליה מעבירים את תוכנו) כעת בפונקציה ()get_detailes הפרמטרים הם מטיפוס מצביע מכיוון ששלחנו את הכתובת של המשתנים pr1,pr2,pr3 אז הפרמטרים של הפונקציה get_detailes צריכים להיות משתני כתובת - כלומר מצביעים

כעת כל שנחוץ לנו הוא להבין כי לכל משתנה יש כתובת בזיכרון ותוכן, ותוכנו של משתנה מטיפוס מצביע הוא לא ערך פשוט אלא כתובת.
נניח כי p1,p2,p3 בעלי הכתובות FF90,FF92,FF94 בהתאמה (גודלו של משתנה כתובת הוא 2 בתים). אזי ערכם יהיה הכתובות של pr1,pr2,pr3 בהתאמה. כדי לגשת לשדות של המשתנים pr1,pr2,pr3 יש להשתמש באופרטור הכוכבית (מכיוון שהקדימות של אופרטור הנקודה עליונה מקדימותו של אופרטור הכוכבית יש לשים סוגריים מסביב לאופרטור הכוכבית כדי שהוא יתבצע ראשון) - כדי לפנות לשדה f_name של pr1 יש לרשום (*p1).f_name שמשמעו ראה מהו) p1* שהוא הכתובת של (pr1 ופנה לשדה f_name שלו (על ידי האופרטור הנקודה).
כעת נראה את התוכנית במלואה:
דוגמא 1:

```
#include <stdio.h>

struct person
{
   char f_name[15];
   char l_name[15];
   int age;
};

void get_detailes (struct person* p1, struct person* p2, struct person* p3)
{
   flushall();
   printf ("Enter first name: ");
   gets ((*p1).f_name);
   printf ("Enter last name: ");
   gets ((*p1).l_name);
   printf ("Enter age: ");
   scanf ("%d", &(*p1).age);

   flushall();
   printf ("Enter first name: ");
   gets ((*p2).f_name);
   printf ("Enter last name: ");
   gets ((*p2).l_name);
   printf ("Enter age: ");
   scanf ("%d", &(*p2).age);
```

```
    flushall();
    printf ("Enter first name: ");
    gets ((*p3).f_name);
    printf ("Enter last name: ");
    gets ((*p3).l_name);
    printf ("Enter age: ");
    scanf ("%d", &(*p3).age);
}

void print_detailes (struct person p1, struct person p2,struct person p3)
{
    printf ("\n\rThe detailes of the tree persons: \n\r");
    printf ("%s %s %d \n\r", p1.f_name, p1.l_name, p1.age);
    printf ("%s %s %d \n\r", p2.f_name, p2.l_name, p2.age);
    printf ("%s %s %d \n\r", p3.f_name, p3.l_name, p3.age);
}

void main()
{
    struct person pr1, pr2, pr3;

    get_detailes (&pr1, &pr2, &pr3);

    print_detailes (pr1, pr2, pr3);
}
```

מבנה כטיפוס למערך

ניתן ליצור מערך של מבנה בדיוק כמו בהצהרת מערך רגיל רק הטיפוס לא יהיה פשוט כמו int, float וכו' אלא מבנה. לדוגמא:

struct person p[3];

יוצר מערך של שלושה משתנים מטיפוס person.

כדי לגשת לשדותיו יש לפנות קודם לאיבר שאנו מעוניינים בו במערך - לדוגמא, אנו רוצים לבדוק את השדה f_name באיבר הראשון של המערך. נרשוm p[1].f_name :

בס"ד

דוגמא 2:

```
#include <stdio.h>

struct person
{
    char f_name[15];
    char l_name[15];
    int age;
};

void get_detailes (struct person pr[])
{
    int i = 0;

    for ( ; i < 3; i++)
    {
      flushall();
      printf ("%d.Enter first name: ", i+1);
      gets (pr[i].f_name);
      printf ("%d.Enter last name: ", i+1);
      gets (pr[i].l_name);
      printf ("%d.Enter age: ", i+1);
      scanf ("%d", &pr[i].age);
    }

}

void print_detailes (struct person pr[])
{
    int i = 0;
    printf ("\n\rThe detailes of the tree persons: \n\r");

    for ( ; i < 3; i++)
      printf ("%s %s %d \n\r", pr[i].f_name, pr[i].l_name, pr[i].age);
}

void main()
{

    struct person pr[3];

    get_detailes (pr);

    print_detailes (pr);
}
```

בס"ד

מבנה מקונן

שדות המבנה יכולים להיות בעצמם מבנה לדוגמא:

```
struct address
{
   char city[15];
    char street[15];
};

struct citizen
{
    struct address add;
    long id;
};
```

נניח כי אנו מצהירים על משתנה מסוג:citizen
struct citizen ctzn;
כדי לגשת לשדה id נרשום ctzn.id :
כדי לגשת לשדה city נרשום ctzn.add.city
כלומר ראשית ניגש לשדה add שהוא גם מטיפוס מבנה וממנו ניגש לשדות שלו שוב על ידי אופרטור הנקודה.

דוגמא 3:

```
#include <stdio.h>

struct person
{
    char f_name[15];
    char l_name[15];
    int age;
};

struct worker
{
    struct person pr;
    int experience;
};

void get_detailes (struct worker wr[])
{
    int i,j;
    for (i = 0; i < 3; i++)
    {
       printf ("%d.Enter first name: ",i+1);
       gets (wr[i].pr.f_name);
       printf ("%d.Enter last name: ", i+1);
```

```
        gets (wr[i].pr.l_name);
        printf ("%d.Enter age: ", i+1);
        scanf ("%d", &wr[i].pr.age);
        printf ("%d.Enter years of experience: ", i+1);
        scanf ("%d", &wr[i].experience);
    }
}

void print_detailes (struct worker wr[])
{
    int i;

    printf ("\n\rThe detailes of the tree persons: ");
    for (i = 0; i < 3; i++)
    {
        printf ("\n\r%s %s ", wr[i].pr.f_name, wr[i].pr.l_name);
        printf ("%d %d ", wr[i].pr.age, wr[i].experience);
    }
}

void main()
{

    struct worker wr[3];
    get_detailes (wr);
    print_detailes (wr);

}
```

הערות:
ניתן לבנות מבנים בעלי קינון מדרגה גבוהה יותר (כלומר שדה של שדה של מבנה וכו').
במקרה ונרצה לשמור את הנתונים של המבנה בקובץ יש להעתיק לתוך הקובץ את המשתנים של המבנה.

שמירת מערך מטיפוס מבנה בקובץ

כדי לבצע זאת יש לפתוח את הקובץ לקריאה לאחר שפעולה זו הסתיימה בהצלחה יש לעבור כל כל אברי המערך בלולאה ולהעתיק איבר איבר ושדה שדה לקובץ כך:

```
for ( ; i < LENG; i++)
{
    fprintf (f, "%s %s ", wr[i].pr.f_name, wr[i].pr.l_name);
    fprintf (f, "%d %d\n\r", wr[i].pr.age, wr[i].experience);
}
```

נניח שהמערך שלנו הוא מערך של עובדים ושמו wr בעל השדות f_name, l_name, age, experience

שהם שם פרטי,שם משפחה,גיל וניסיון בהתאמה. נדפיס לקובץ שהמצביע שלו הוא f בתחילה את השדות של העובד הראשון wr[0] נתקדם בלולאה לאיבר הלאה וכך הלאה עד שהגענו לאיבר האחרון wr[LENG-1] כי תנאי הלולאה הוא. i < LENG
כעת נרשום את התוכנית במלואה:

דוגמא 4:

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

#define LENG 3

struct person
{
   char f_name[15];
   char l_name[15];
   int age;
};

struct worker
{
   struct person pr;
   int experience;
};

void get_detailes (struct worker wr[])
{
   int i;
   for (i = 0; i < LENG; i++)
   {
      flushall();
      printf ("%d.Enter first name: ",i+1);
      gets (wr[i].pr.f_name);
      printf ("%d.Enter last name: ", i+1);
      gets (wr[i].pr.l_name);
      printf ("%d.Enter age: ", i+1);
      scanf ("%d", &wr[i].pr.age);
      printf ("%d.Enter years of experience: ", i+1);
      flushall();
      scanf ("%d", &wr[i].experience);
   }
}

void print_detailes (struct worker wr[])
{
   int i;
   printf ("\n\rThe detailes of the persons: ");
```

```
    for (i = 0; i < LENG; i++)
    {
       printf ("\n\r%s %s ", wr[i].pr.f_name, wr[i].pr.l_name);
       printf ("%d %d ", wr[i].pr.age, wr[i].experience);
    }
}

void save (struct worker wr[], FILE* f)
{
    char file_name[15];
    int i = 0;

    printf ("\n\rEnter file name to create: ");
    flushall();
    gets (file_name);

    if ((f = fopen (file_name, "wt")) == NULL)
    {
       printf ("\n\rCan't open the file %s ", file_name);
       exit(1);
    }
    else
    {
       printf ("\n\rSaving the data in the file %s ", file_name);

       for ( ; i < LENG; i++)
       {
       fprintf (f, "%s %s ", wr[i].pr.f_name, wr[i].pr.l_name);
       fprintf (f, "%d %d\n\r", wr[i].pr.age, wr[i].experience);
       }
    }
    fclose(f);
    printf ("\n\rThe file has been closed");
}

void main()
{
    FILE * s_f; //sutruct file
    struct worker wr[LENG];

    get_detailes (wr);
    print_detailes (wr);
    save (wr,s_f);
}
```

מבנה כערך המוחזר על ידי פונקציה

כאשר אנו מעוניינים לקלוט נתוני מבנה אנו נעשה זאת בעזרת פונקציה המחזירה ערך מבנה.
בהצהרה על הפונקציה יש צורך לרשום:

תבנית לפונקציה המחזירה ערך מבנה:
struct name_of struct name_of_function(Arguments);

דוגמא 5:

בפונקצית main יש לולאה הקוראת לפונקציה choise המחזירה תו. תו זה נשלח כארגומנט לפונקציה menu הקוראת לשאר הפונקציות על פי בחירת המשתמש. כל פונקציה שנקראה מבצעת את תפקידה ושוב חוזרים לפונקצית main על מנת לאפשר למשתמש לבצע מספר פעולות בתוכנית אחת. ניתן לצאת מהתוכנית על ידי הקשת. Q

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

#define LENG 10
int leng = 0;

struct person
{
    char f_name[15];
    char l_name[15];
    long id;
};

struct student
{
    struct person pr;
    int grade;
};
struct student get_details()
{
    struct student st;

    flushall(); //we need to clean the buffer evry time we
      //get data from keyboard
    printf ("\n\rEnter first name: ");
    gets (st.pr.f_name);
    printf ("\n\rEnter last name: ");
    gets (st.pr.l_name);
    printf ("\n\rEnter id: ");
    flushall();
    scanf ("%ld", &st.pr.id);
    printf ("\n\rEnter grade: ");
```

```
    flushall();
    scanf ("%d", &st.grade);
    return st; //return the student to the fun add_student
}
void add_student (struct student st[],struct student s)
{
    strcpy(st[leng].pr.f_name, s.pr.f_name);
    strcpy(st[leng].pr.l_name, s.pr.l_name);
    st[leng].pr.id = s.pr.id;
    st[leng].grade = s.grade;
    leng++; // update the leng of the array
}
void print_detailes (struct student st[])
{
    int i;
    printf ("\n\rThe detailes of the persons: ");
    for (i = 0; i < leng; i++)
    {
       printf ("\n\r%s %s ", st[i].pr.f_name, st[i].pr.l_name);
       printf ("%ld %d ", st[i].pr.id, st[i].grade);
    }
}

void save (struct student st[], FILE* f)
{
    int i;
    printf ("\n\rThe data will be saved in the file student.txt ");
    flushall();

    if ((f = fopen ("student.txt", "wt")) == NULL)
    {
       printf ("\n\rCan't open the file student.txt ");
       exit(1);
    }
    else
    {
       for (i = 0 ; i < leng; i++)
       {
          fprintf (f, "%s %s ", st[i].pr.f_name, st[i].pr.l_name);
          fprintf (f, "%ld %d\n\r", st[i].pr.id, st[i].grade);
       }
    }
    fclose(f);
    printf ("\n\rThe file has been closed");
}

char choise()
{
    printf ("\n\n\r(A) - add student\n\r");
```

```
    printf ("(W) - write data in to file\n\r");
    printf ("(P) - print to screen\n\r");
    printf ("(Q) - quit\n\r");
    flushall();
    return getchar();
}
void quit()
{
    printf ("\n\rEnd of program");
    exit(1);
}

void menu (struct student st[], FILE *f, char c)
{
    switch (c)
    {
       case 'A': add_student (st, get_details());break;
       case 'W': save (st, f); break;
       case 'P': print_detailes (st); break;
       case 'Q': quit(); break;
       default : choise();
    }
}

void main()
{
    char c;
    FILE * f;
    struct student st[LENG];

    printf ("choose char to: \n\r");

    while (c = choise())
    {
       menu(st, f, c);
    }
}
```

בס"ד

רקורסיה

רקורסיה הוא תהליך בו הפונקציה קוראת לעצמה.
מתכנתים מתקשים להבין את תהליך הרקורסיה ומופתעים מתוצאותיו.
זהו כלי חשוב מאוד וישנם בעיות שהפיתרון הטבעי להן הוא רקורסיה.
נראה את שני סוגי הרקורסיות: נפתח ברקורסיה לא אמיתית הנקראת רקורסית קצה בו הקריאה לפונקציה נעשית בשורה האחרונה והסיבה היא פשוטה ובסה"כ מקבלים לולאה פשוטה (נראה זאת בהמשך על ידי דוגמא) והשנייה רקורסיה אמיתית שהקריאה מופיעה בגוף הפונקציה.

דוגמא 1:

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

void s_recursion (int n) //s_recursion
{
   if (n > 0)
   {
      printf ("%d \n\r", n);
      s_recursion (n-1);
   }
}

void main()
{
   int n = 10;
   s_recursion (n);
}
```

דוגמא לרקורסיית קצה, כפי שאמרנו רקורסיה זו זהה ללולאה פשוטה.

דוגמא 2:

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

void r_recursion (int n) //real recursion
{
   if (n > 0)
   {
      r_recursion (n-1);
      printf ("%d \n\r", n);
   }
}

void main()
{
   int n = 10;
```

```
    r_recursion (n);
}
```

קוראת לעצמה בתחילת הפונקציה. הפלט r_recursion זוהי דוגמא לרקורסיה אמיתית. הפונקציה הוא 1 עד 10 והסיבה היא שהתוכנית לא נעצרת בתנאי העצירה אלא רק באחד מעותקיה .
כשפונקציה קוראת לעצמה בעצם נוצר עותק של הפונקציה ובתוכנית שלנו נוצרו 10 עותקים.
כשהאחרון מבניהם לא מקיים את תנאי העצירה הוא נסגר וכל שאר העותקים ממשיכים להתבצע במלואם מהנקודה שיצאנו מהם (במקרה שלנו משורת ההדפסה) ולכן המספרים מודפסים בסדרם ההפוך.

נסכם ונאמר: אם הקריאה לפונקציה מתבצעת באמצע הפונקציה אז לאחר שהגענו לתנאי הקצה ואנו בעותק האחרון, נסיים לבצע אותו ונחזור לעותק לפני אחרון (זה שקרא לאחרון) לשורה הבאה שלא הספקנו לבצע (עקב הקריאה לעותק הבא אחריה שהוא היה העותק האחרון) וכך נסיים אותו. משם נלך לעותק הלפני לפני, נסיים את הפקודות שלא ביצענו וכך הלאה עד אשר נגיע לעותק הראשון ונמשיך אותו מהשורה הראשונה שלא ביצענו וכך נסיים את פעולת הרקורסיה.

בעצם יש מספר תכונות לפונקציה רקורסיבית:
1. **תנאי קצה**- כדי שהרקורסיה תיעצר בקריאה מסוימת. זהו תנאי חשוב ביותר - אם לא נדאג לכך נקבל לולאה אינסופית.
2. **קריאה לפונקציה עצמה**(בדר"כ עם ערך קטן או גדול יותר).

מניחים קיום פיתרון n כך למעשה ניתן לפתור בעיות רבות בעזרת רקורסיה: נתונה בעיה מסדר n כותבים תנאי עצירה ועל סמך אמונה של פיתרון בעיה ל , n-1 כותבים את n-1 מסדר הפונקציה.

לדוגמא:
בניית הפונקציה factorial(n): - n חישוב עצרת של מספר
כתיבת תנאי עצירה: if (n = 0) return 1
כלומר 0! שווה ל 1,
factorial(n-1) כלומר n-1 מניחים קיום פתרון של בעיית העצרת של ,
שלב אחרון: פתרון הבעיה:
facroaial(n) = factorial(n-1)*n
נממש את הפיתרון:

דוגמא 3:

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

long factorial (int n)
{
    if (n > 0)
      return (n * factorial (n-1));
    else
      return 1;
```

```
}
void main()
{
   int n = 5;
   printf ("The factorial of %d is: %ld", n, factorial(n));
}
```

נוצרו 5 עותקים, האחרון לא מקיים את תנאי הקצה ולכן הערך המוחזר הוא 1. העותק האחרון עותק זה מסתיים התוצאה (1), (1) nהסתיים, העותק שלפניו מקבל 1 ומכפיל אותו בערכו של עותק זה מסתיים והתוצאה (2) נשלחת, (2) n מוחזרת לעותק שנוצר לפני ושוב מכפיל בערכו של כעת התוצאה 6 וכך הלאה עד לסיימו של העותק הראשון (3) n לעותק שלפניו, מוכפלת בערכו של.

דוגמא 4:

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

long fib (int n)
{
   if ((n == 0) || (n ==1))
      return 1;
   else
      return (fib (n-1) + fib (n-2));
}

void main()
{
   int n = 0;
   for ( ; n <= 20; n++)
      printf ("Fibunachi (%d): %ld\n\r", n, fib(n));
}
```

התוכנית מחשבת את סדרת פיבונצ'י עד האיבר ה 20 שחוקיותה: שני האיברים הראשונים שווים ל-1. וכל שאר אברי הסדרה שווים לחיבור שני האיברים שלפניהם.

דוגמא 4.1:

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

long fib (int n)
{
   if ((n == 0) || (n == 1))
      return 1;
   else
      return (fib (n-1) + fib (n-2));
```

```
}

long d_fib (int n)
{
   if ((n == 0) || (n == 1))
      return 0;
   else
      return (fib (n) - fib (n-1));
}

void main()
{
   int n;
   for (n = 0; n <= 20; n++)
      printf ("Fibunachi (%d): %ld\n\r", n, fib(n));
   for (n = 2; n <= 22; n++)
      printf ("Diffrance_Fib (%d): %ld\n\r", n, d_fib(n));
}
```

הרחבה של התוכנית הקודמת - נבנה את סדרת ההפרשים בין שני אברי עוקבים של סדרת פיבונצ'י.
יש לשים לב כי גם היא מהווה סדרת פיבונצ'י מהאיבר השני והלאה.
סדרה זו צופנת בחובה המון יופי ומורכבות - למתעניינים יש מספר רב של ספרים החוקרים את הסדרה המיוחדת הזו.

הדפסת קובץ הפוך:

```
void reverse (FILE *f)
{
   char ch;
   static temp = 0;
   if ((fscanf (f, "%c", &ch)) != EOF)
      reverse(f);
   if (temp)
      printf ("%c", ch);
   temp = 1;
}
```

הפונקציה מקבלת מצביע לקובץ המבוקש. בפונקצית ההדפסה השתמשנו במשתנה סטטי - temp יחודו הוא בכך שהוא שומר על ערכו בפונקציה אפילו אם יוצאים מהפונקציה, ובפעם הבאה שנשתמש בפונקציה ערכו לא יאתחל אלא יהיה הערך האחרון שעודכן. השתמשנו במשתנה מיוחד זה מכיוון שאנו רוצים לעבור על כל הקובץ וברגע שנגיע לסוף הקובץ נדפיס את תוכנו מהסוף מכיל תו אחר -ביציאה ch להתחלה. לא יהיה בכך קושי מכיוון שבכל אחד מהעותקים המשתנה מהרקורסיה אנו נדפיס את התו האחרון ולאחר מכן את התו הלפני אחרון עד שנגיע לעותק הראשון ונדפיס את התו הראשון שקראנו.
הבעיה היא שמגיעים לתו סוף קובץ ולא מעוניינים להדפיסו ולכן המשתנה הסטטי ערכו 0 וכשמגיעים לא מודפס ומשנים את ערכו של המשתנה הסטטי ל 1 וכעת כל שאר (לסוף קובץ תו זה EOF) התווים של הקובץ יודפסו למסך.

כעת נראה את התוכנית במלואה:

דוגמא 5:

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

void reverse (FILE *f)
{
   char ch;
   static temp = 0;
   if ((fscanf (f, "%c", &ch)) != EOF)
      reverse(f);
   if (temp)
      printf ("%c", ch);
   temp = 1;
}

void main()
{
   FILE *f;
   char file_name[20];
   printf ("Enter a file name to show reverse: ");
   scanf ("%s", file_name);
   if ((f = fopen (file_name, "rt")) != NULL)
      reverse (f);
   else
   {
      printf("\n\rCan't open file %s ", file_name);
      exit(1);
   }
}
```

דוגמא 6:

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

int paskal (int i, int j)
{
   if (j == 0 || j == i)
      return 1;
   else
      return paskal (i-1, j-1)+paskal (i-1, j);
}
```

```
void main()
{
   int i,j;
   int leng = 9;
   for (i = 0; i < leng; i++)
   {
      putchar ('\n');
      for (j = 0; j < i+1; j++)
         printf ("%d ",paskal(i, j));
   }
}
```

בנית משולש פסקל בעזרת רקורסיה. הקוד קצר לעין שיעור ביחס ללולאות במקרים כאלו.

כאמור ניתן לראות את החיסכון האדיר בשורות קוד כאשר משתמשים ברקורסיה על פני לולאות. ברגע שמבינים את הרקורסיה הרבה יותר פשוט לפתור בעיות מורכבות על ידי רקורסיה מאשר מספר רב של לולאות.

אמרנו בתחילה כי ברקורסיה יש קריאות לעותקים מהפונקציה, העתקים אלו נשמרים במחסנית והקריאות משתחררות אחת אחר השנייה מהקריאה העליונה (האחרונה) עד לראשונה ומכיוון שכמות הזיכרון של המחשב מוגבלת יש מספר חסום של קריאות - כלומר אם נקרא לפונקציה יותר משיוכל המחשב לאחסן הוא ייעצר. לדוגמא אם נרצה לחשב את האיבר ה- 100 של סדרת פיבונצ'י סביר להניח שלא נוכל לעשות זאת (במחשב רגיל). לכן לא תמיד עדיף להשתמש ברקורסיה ויש לזכור שמספר הקריאות חסום.

ניתן לייצר רקורסיות מורכבות יותר על ידי כך שתהיה יותר מקריאה אחת לפונקציה בפונקציה. יש לזכור שכאשר עותק מסתיים חוזרים לעותק שלפניו מהמקום שיצאנו ממנו.

דוגמא 7:

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

void func1 (int n)
{
   if (n > 0)
   {
      gotoxy (4*n, wherey());
      printf ("Enter no_%d\n", n);
      func1 (n-1);

      gotoxy (4*n, wherey());
      printf ("Exit no_%d\n", n);
      func1 (n-1);
```

```
    }
}

void func2 (int n)
{
    if (n > 0)
    {
       gotoxy (4*n, wherey());
       printf ("Enter no_%d\n", n);
       func2 (n-1);
       gotoxy (4*n, wherey());
       printf ("Exit no_%d\n", n);
       func2 (n-1);
    }
}
void func3 (int n)
{
    if (n > 0)
    {
       func3 (n-1);
       gotoxy (4*n, wherey());
       printf ("Enter no_%d\n", n);

       gotoxy (4*n, wherey());
       printf ("Exit no_%d\n", n);
       func3 (n-1);
    }
}

void func4 (int n)
{
    if (n > 0)
    {
       func4 (n-1);
       gotoxy (4*n, wherey());
       printf ("Enter no_%d\n", n);

       func4 (n-1);
       gotoxy (4*n, wherey());
       printf ("Exit no_%d\n", n);
    }
}

void main()
{
    int num = 3;
    clrscr();

    func1 (num);
```

```
    printf("\n\n");
    func2 (num);

    getchar();
    clrscr();

    func3 (num);
    printf("\n\n");
    func4 (num);
    getchar();
}
```

השתמשנו הפונקציה getchar על מנת שנוכל לחלק את הפלט לשני חלקים ונוכל לראות זאת על ידי השהית הפלט עד לקליטת מקש.
הצעה: רשום את ניחושיך בנוגע לפלט של התוכנית וראה אם צדקת
בתוכנית ביצענו שימוש בפונקציה gotoxy על מנת שיהיה ברור באיזה שלב ברקורסיה אנו נמצאים.
ככל שהכיתוב עמוק יותר) כלומר שדה ה x גדול יותר כך אנו יותר "עמוק" ברקורסיה וההיפך.(

חשוב תמיד לזכור שאפילו אם יש מספר קריאות לפונקציה ברגע שאנו חוזרים לעותקים הראשוניים אנו ממשיכים מהשורה הראשונה שהפסקנו לבצע את הפקודות באותה קריאה של הפונקציה ואף אם בהמשך יש שוב קריאה לפונקציה אנו נכנס שוב לעותקים הבאים ובקיפול (חזרה) של הרקורסיה נמשיך שוב מהשורה הראשונה שלא ביצענו בפונקציה.

בס"ד

מצביעים (pointers)

מצביע הינו משתנה המכיל נתון מסוג כתובת, זהו טיפוס נתונים המאפשר פניה לכתובת של מקום בזיכרון.
כאשר מצהירים על משתנה מסוג מצביע יש להשתמש באופרטור הכוכבית (*) המציין כי המשתנה הוא מסוג מצביע.

לדוגמא:
int *pi- בהצהרה זו אנו אומרים 2 דברים: הראשון כי המשתנה pi הינו משתנה מסוג מצביע לשלם, והשני (pi*) הוא הערך המוצבע על ידי pi הוא שלם.
כלומר אם נסתכל בתוכנו של pi נראה כי מכיל כתובת לזיכרון, ו (pi*) הוא התוכן של הכתובת שמוכלת ב pi .

ניתן ליצור משתנים מסוג מצביע לכל טיפוסי הנתונים שהזכרנו עד כה: מצביע לשלם, לממשי וכו' ואף לטיפוסים מורכבים יותר: למערך ומבנה.

משתנה מסוג מצביע תופס 2 בתים בזיכרון (לשם אחסון ערך כתובת).
ניתן לבצע הקצאת זיכרון על ידי פקודת malloc (הקצאת זיכרון דינאמית - נסביר בהמשך ביתר פירוט) , למשל מעוניינים לבצע הקצאת זיכרון למשתנה pi נרשום:
pi = malloc(2);

פקודה זו מחזירה ערך של כתובת של בלוק בגודל 2 בתים (גודל של int).
בלוק הזיכרון מוקצה מתוך אזור זיכרון הנקרא ערימה (heap).

נניח כי הכתובת המוחזרת על ידי פקודת new היא FFF4 והתוכן של כתובת זו הוא 14, אזי ערכו של pi הוא FFF4 וערכו של (pi*) הוא 14.
כשאנו נרצה לקבל הקצאת זיכרון אנו נרצה לשנות את הערך של הכתובת שקיבלנו (שכן ערכה חסרת משמעות לגבינו) נעשה זאת על ידי הצבה בערך המוצבע כך:

*pi = 20 - פנינו לכתובת של p ושינינו את ערכה.

בפרקים קודמים הסברנו כי האופרטור & מציין כתובת של משתנה ולכן:
אם נרצה לפנות לכתובת של pi נרשום: &pi .
אם נרצה לפנות לערכו נרשום: pi.
אם נרצה לפנות לערך המוצבע נרשום: pi*.

לא ניתן לבצע הצבת כתובת לאחר הצהרת על משתנה מצביע כך:

```
int *pi;
pi = FFF4; (לא תקין).
```

אך אם יש בידינו משנה שכבר הוקצה בעבורו מקום בזיכרון ניתן להציב את כתובתו במצביע כך:

```
int x;
int *pi;
pi = &x;
```

אם הכתובת של x היא B206 אזי pi יכיל את כתובת זו.

זיכרון דינמי

ישנן שתי שיטות לביצוע הקצאת זיכרון: הקצאת זיכרון סטטית והקצאת זיכרון דינאמית.

הקצאת זיכרון סטטית -המהדר קובע את דרישות האחסון על פי הצהרת המשתנים (כמובן בזמן ההידור).
הקצאת זיכרון דינאמית -הקצאת מקום נעשה בזמן ביצוע התוכנית על יד פקודת malloc

מכיוון שהקצאת הזיכרון נעשתה בזמן ביצוע התוכנית יש לדאוג לשחרר את הזיכרון ,נעשה זאת על ידי פקודת.free (name_pointer)

חשוב מאוד לשחרר את הזיכרון בעבור משתנים שההקצאה עבורם הייתה דינאמית, אם לא נעשה זאת המקום בזיכרון יהיה תפוס ולא נוכל לשנות ולהשתמש בו שוב. מצב זה נקרא דליפת זיכרון - במצב זה מוקצה זיכרון שוב ושוב ואם אין משחררים מגיעים למצב שאין עוד מקום בזיכרון להקצאות נוספות הנדרשות על ידי התוכנית. פונקציה זו נמצאת בספרייה .alloc.h
משתנה שבעבורו הייתה הקצאת זיכרון סטטית המהדר דואג לשחרור הזיכרון.

נסכם את מה שהסברנו עד כה: מצביע הינו משתנה שמצביע למשתנה אחר, משמע שהמצביע תוכנו הוא כתובת של משתנה אחר. המצביע לא מכיל ערך במובן המקובל אלא מכיל כתובת של משתנה המכיל ערך סטנדרטי.
מכיוון שמצביע מכיל כתובת ולא ערך הוא מחולק ל- 2: המצביע המכיל כתובת וכתובת מוצבעת המכילה ערך.

נושא המצביעים הוא אחד הנושאים הכי קשים בשפה ועם זאת בעל כוח רב - לדוגמא כשאנו מעוניינים לבנות רשימה של סטודנטים, אין אנו יודעים כמה מקומות להקצות כי יש סטודנטים והרשימה גדלה ולכן פה יש שימוש במצביעים - מכיוון שאין אנו צריכים לדעת מה מספר הסטודנטים אלא אנו יכולים לצרף סטודנטים חדשים לרשימה על ידי שימוש ברשימה מקושרת שהיא מעיין מערך דינאמי. יש בכך חיסכון גדול מכיוון שאם הינו עובדים עם מערך הינו צריכים להגדיר את גודלו ואם באיזשהו שלב היו נגמרים המקומות במערך אז אנו בבעיה וכן בהתחלה מכיוון שידוע כי יתווספו עד סטודנטים לרשימה אנו מקצים שטח זיכרון רב מדי (שכרגע הוא מבוזבז) - כל זאת נמנע אם משתמשים במצביעים.

הערה:
בפונקצית ההדפסה יש להשתמש בתבנית ההמרה p% בכדי להדפיס כתובת.

דוגמא 1:

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

```
void main()
{
    int i, j;
    int *pi; // a pointer to an integer

    printf("The address of i is: %p\n\r",&i);
    pi = &i;
    printf("The value of pi is: %p\n\r", pi);
    *pi = 5;
    printf("The value of i is: %d\n\r", i);
    j = i;
    printf("The value of j is: %d\n\r", j);
    printf("The value of *pi is: %d", *pi);
}
```

הערה: כשמצהירים על מצביע p אין הוא מצביע על ערך "חוקי", חובה להציב ערך כתובת לפני שמשתמשים בערך המוצבע .(*p)
לדוגמא:

```
int *pi;
*pi = 100; (לא חוקי)
```

פעולות אלו ייצרו טעויות העלולות לגרום לקריסת התוכנית.
הדרך הנכונה היא:

```
int *pi;
int x;
pi = &x;
*pi = 5;
```

דוגמא 2:

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

void main()
```

```
{
    int i = 10;
    int *pi;

    printf ("&i= %p i= %d \n\r", &i, i);
    printf ("&pi= %p pi= %p *pi= %d\n\r", &pi, pi, *pi);

    pi = &i;
    printf ("&pi= %p pi= %p *pi= %d\n\r", &pi, pi, *pi);

    (*pi)++;
    printf ("&i= %p i= %d\n\r", &i, i);
    printf ("&pi= %p pi= %p *pi= %d\n\r", &pi, pi, *pi);
    pi = (int*)malloc(sizeof(int*));
    *pi = 100;
    printf ("&i= %p i= %d\n\r", &i, i);
    printf ("&pi= %p pi= %p *pi= %d", &pi, pi, *pi);

    free(pi);
}
```

מצביע כפרמטר של פונקציה

נפתח בדוגמא:

דוגמא 3:

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

void swap (int *pi, int *pj)
{
    int temp;
    printf ("&ip= %p pi= %p *pi= %d\n\r", &pi, pi, *pi);
    printf ("&jp= %p pj= %p *pj= %d\n\r", &pj, pj, *pj);

    temp = *pi;
    *pi = *pj;

    *pj = temp;

    printf ("\n\rAfter swapping there values:\n\n\r");
    printf ("&ip= %p pi= %p *pi= %d\n\r", &pi, pi, *pi);
    printf ("&jp= %p pj= %p *pj= %d\n\r", &pj, pj, *pj);
}
```

```
void main()
{
   int i = 10;
   int j = 20;

   swap (&i, &j);

}
```

הערה לדוגמא :אם הפרמטרים של פונקצית swap לא היו מצביעים לא היינו מקבלים את אותה התוצאה: הערכים (בלבד) שלi ו j -היו נשלחים לפונקציה וערכי i ו j-המקוריים היו נשארים כמו שהם.

הערה:
מומלץ לעקוב אחר כל הכתובות של המשתנים ואחר השינויים.

הערה:
כאשר קוראים לפונקציה עם ארגומנטים מסוימים, המהדר מקצה זיכרון לפרמטרים של הפונקציה. זיכרון זה משתחרר כאשר הפונקציה מסתיימת, ולכן אם נתבונן בכתובת של הפרמטרים נראה שהם שונים מהכתובות של הארגומנטים ששלחנו, אך אם שולחים לפי כתובת (&) הכתובות של ערכי הארגומנטים והפרמטרים שווים וזאת מכיוון שאין המהדר מקצה זיכרון לפרמטרים אלא משתמש בכתובות של הארגומנטים ששלחנו.
אין זה פלא שכאשר קוראים למספר פונקציות ערכי הפרמטרים זהים וזאת מכיוון ששוחרר הזיכרון בסיומה של כל פונקציה וכעת ניתן להשתמש באותו זיכרון בעבור הפונקציות הבאות.

דוגמא 4:

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

void fun1 (int *p)
{
   printf ("In fun1: &p= %p p= %p *p= %d\n\r", &p, p, *p);
}

void fun2 (int *p)
{
   printf ("In fun2: &p= %p p= %p *p= %d\n\r", &p, p, *p);
}

void fun3 (int p)
{
   printf ("In fun3: &p= %p p= %d\n\r", &p, p);
}

void main()
```

```
{
    int *p = (int*)malloc (sizeof(int));
    *p = 100;
    printf ("In main: &p= %p p= %p *p= %d\n\r", &p, p, *p);
    fun1(p);
    fun2(& *p);
    fun3(*p);

    free(p);
}
```

פונקציות fun1 ו fun2 זהות מכיוון ש *p & משמעו הכתובת של המשתנה *p וזהו בדיוק p
כל השאר מתקיים כפי שהסברנו למעלה.

הרחבה של הדוגמא הקודמת:

דוגמא 5:

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

void fun1 (int *p)
{
    int i = 10;
    p = &i;
    printf ("In fun1: &p= %p p= %p *p= %d\n\r", &p, p, *p);
}

void fun2 (int *p)
{
    *p = 1;
    printf ("In fun2: &p= %p p= %p *p= %d\n\r", &p, p, *p);
}

void fun3 (int p)
{
    p = 2;
    printf ("In fun3: &p= %p p= %d\n\r", &p, p);
}

void fun4 (int* p)
{
    int i = 5;
    *p = i;
    printf ("In fun2: &p= %p p= %p *p= %d\n\r", &p, p, *p);
}
```

```
void main()
{
    int *p = (int*)malloc(sizeof(int));
    *p = 8;
    printf ("In main: &p= %p p= %p *p= %d\n\r", &p, p, *p);
    fun1(p);
    fun2(& *p);
    fun3(*p);
    fun4(&*p);
    printf ("In main: &p= %p p= %p *p= %d\n\r", &p, p, *p);
    free(p);
}
```

כתובת של מערך

כשדנו במערכים טענו כי המהדר מקצה זיכרון לבלוק זיכרון רציף בהתאם לגודלו של המערך (והסברנו גם איך מחשבים את הכתובת של כל איבר בזיכרון). כעת ברור כי מערך הוא בעצם מצביע, וזאת מכיוון שידוע לנו הכתובת של המערך (הכתובת של האיבר בראשון) וכל שאר האיברים נגישים לפי ההיסט מהכתובת של האיבר הראשון.בעצם המצביע (המערך) מכיל את הכתובת הראשונה (האיבר הראשון). כאשר המערך הוא של שלמים המצביע יהיה מסוג שלם ואם המערך הוא של ממשים אז גם המצביע יהיה מסוג ממשי.
לדוגמא:
נצהיר על מערך של שלמים בגודל 5

int arr_i[5];

arr_i	2000	2002	2004	2006	2008
2000	arr_i[4]	arr_i[3]	arr_i[2]	arr_i[1]	arr_i[0]

כל התאים הם מסוג שלם , שם המערך arr_i יהיה משתנה מצביע מסוג שלם ומכיל את הכתובת של האיבר הראשון במערך.(2000)

ניתן לגשת לאיבר הראשון במערך באופן הבא:

```
*(arr_i)
ולאיבר השני:
*(arr_i+1)
לאיבר האחרון:
*(arr_i+4)
נכליל זאת, על מנת לגשת לכתובת של a[i]:
*(a+i-1).
```

הסבר:
כשאר כותבים arr_i+1 הכוונה לכתובת הבאה אחרי הכתובת של arr_i ומכיוון שהמערך שלפנינו הוא מסוג שלם הקפיצות יהיו של שני בתים (בניגוד לאינטואיציה הקובעת שהקפיצות יהיו בבית אחד). העניין נעוץ באריתמטיקה של מצביעים: לדוגמא אם p הוא מצביע מסוג ממשי אזי הפעולה ++pתקדם את p ב 4 בתים וזאת מכיוון שגודלו בזיכרון של ממשי הוא 4.
ניתן לבצע גם הגדלה מאוחרת ומוקדמת של מצביעים , חיסור וחיבור בשלם

דוגמא 6:

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

void main()
{
   int i = 0;
   float arr_f[] = {1.1, 2.2, 3.3, 4.4, 5.5};

   for ( ; i < 5; i++)
      printf ("array[%d] = %f\n\r",i , *(arr_f+i));
}
```

דוגמא 7:

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

void main()
{

   char *string = "Hello Student";

   puts (string+1);

   puts (++string);

   puts (string-1);
   puts (string+2);
}
```

פקודתtypedef

ניתן להגדיר מחדש כל טיפוס על ידי פקודת. typedef

בס"ד

תבנית:
typedef character new_character;

יחליף את הטיפוס characterבטיפוס החדש - new_charecter כלומר אם נגדיר משתנה מהטיפוס החדש הוא יתייחס אליו כמו בהתייחסותו אל משתנה שהוגדר כטיפוס הישן. לדוגמא:

```
typedef int* pint;
pint p;
```

משמע p :הוא משתנה מצביע לשלם מכיוון שהגדרנו מצביע לשלם כ -pint -המהדר מחליף כל מופע של pint ב. int*

האופרטורsizeof

האופרטור sizeof מחזיר את גודלו של הטיפוס בבתים.

תבנית:
sizeof (name_of_charecter) לדוגמא:
sizeof (float) מחזיר 4

אילו הינו כותבים:

```
float f
sizeof(f) - (4) הינו מקבלים אותו פלט .
```

הכללת הקובץ:alloc.h
ספרייה זו מכילה פונקציות בעבור פעולות של הקצאה ושחרור זיכרון.
כבר דיברנו על הפקודה free לשחרור זיכרון שהוקצה באופן דינאמי.

הקצאה דינמיתmalloc -

כעת נכיר את הפונקציה להקצאת זיכרון בזמן ביצוע - malloc לשם כך יש צורך להגדיר מצביע למערך ולציין את גודל המערך בבתים.
כפי שהסברנו מצביע למערך יכיל את הכתובת של האיבר הראשון במערך.
גודלו של המערך הוא: גודל הטיפוס*מספר איברים במערך לכן מערך של שלמים בגודל 5 יכיל 10 בתים (כי כל שלם תופס 2 בתים).

התבנית לשימוש בפונקציה :malloc()
variable_pointer = (pointer_type) malloc (size_of_memory);

דוגמא:

```
int size, *p_list;
printf("Enter the number of elements:");
```

```
scanf("%d", &size);
p_list = (int*)malloc (size * sizeof(int));
```

הקצאת זיכרון בעזרת הפונקציהcalloc

בפונקציה malloc יש צורך לחשב את גודל מקום בזיכרון. בפונקציה calloc אין צורך לעשות זאת אלא רק לשלוח כארגומנטים לפונקציה מספר האיברים הדרוש וכארגומנט השני גודל כל איבר.

התבנית לשימוש בפונקציה :calloc()

```
variable_pointer = (casting) calloc (num_of_elements, size_of_element);
```

יש צורך ב casting לפני הפקודה calloc כמו בפונקציה malloc לפי סוג הטיפוס שבעבורו מקצים זיכרון.

דוגמא:

```
int size, *arr_p;
printf("Enter the number of elements:");
scanf("%d", &size);
arr_p = (int*)calloc (size, sizeof(int));
```

הערה:
כאשר מקצים זיכרון הערך המוחזר על ידי פונקצית הקצאת זיכרון היא כתובת.
יש מקרים שבעבורם הקצאת הזיכרון נכשלת והערך המוחזר על ידי הפונקציה היא null ומכיוון שלא קיבלנו כתובת לא ניתן להשתמש במצביע כמצביע "חוקי" ונרצה לסיים את התוכנית. לכן לאחר הקצאת זיכרון תמיד צריך לבדוק אם ההקצאה הצליחה.

דוגמא 8:

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

void main()
{
   long *l_list;
   l_list =(long*) malloc (5*sizeof(long));
   if (l_list == NULL)
   {
      printf ("Failed to allocate memory");
      return;
   }
   printf ("The address of l_list is: %p", &l_list);
```

```
    free(l_list);
}
```

שינוי גודל מערך דינאמי בזמן ביצוע

מערך דינאמי הוא בעצם מערך שניתן לשנות את גודלו. מצהירים על מצביע בגודל מסוים (מספר איברים כרצוני) על ידי הפונקציות שהזכרנו malloc או calloc, כעת נרצה לשנות את גודלו של המערך בזמן ביצוע התוכנית לשל כך נשתמש בפונקציה.realloc

התבנית לשימוש בפונקציה :()realloc

variable_pointer = (casting) realloc (variable_pointer, size_of_memory);

דוגמא 9:

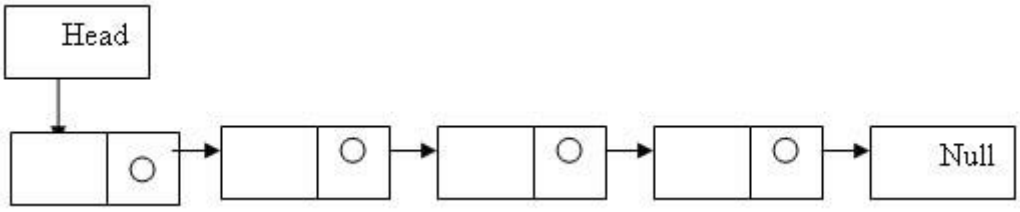
```
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>
#include <conio.h>

void main()
{
    int i, leng = 5;
    int *i_arr = (int*) calloc (leng, sizeof(int));

    if (i_arr == NULL)
    {
      printf ("Failed to allocate memory");
      return;
    }

    for (i = 0; i < leng; i++)
      i_arr[i] = i*i;

    for(i = 0; i < leng; i++)
      printf("%d ", i_arr[i]);

    leng += leng;
    i_arr = (int*)realloc (i_arr, sizeof(int));

    if (i_arr == NULL)
    {
      printf ("Failed to allocate memory");
      return;
    }
    printf("\n\rChanging the size of i_arr to %d:\n\r", leng);
```

```
    for (i = leng/2; i < leng; i++)
      i_arr[i] = i*i;

    printf("The num i_arr is:\n\r");
    for(i = 0; i < leng; i++)
      printf("%d ", i_arr[i]);
}
```

רשימות מקושרות

בפרק הקודם הצגנו מצביעים.
טיפוס מצביע מאפשר לנו להגדיר מבני נתונים דינאמיים נוספים, כגון רשימה מקושרת.
רשימה מקושרת היא מבנה נתונים בו כל איבר מצביע על האיבר הבא אחריו.
כלומר כל איבר יכיל את השדות ה"רגילים" - מטיפוסים פשוטים שונים (שלם, ממשי וכו') או אף מבנים וכן לפחות שדה אחד מטיפוס מצביע.
היתרון הוא כפי שהזכרנו בפרק בקודם חיסכון בזיכרון ויכולת להגדלה או הקטנה של מבני הנתונים.
כל איבר ברשימה נקרא צומת, כאשר הצומת הראשונה היא ראש הרשימה והאחרונה סוף הרשימה, שלא מצביעה על אף איבר (null מכיל).

סימון:
משתנה מסוג רשימה- מלבן עם עיגול בתוכו.
מלבן - נתונים.
כל איבר ברשימה מכיל מצביע לאיבר הבא ונתונים חוץ מראש הרשימה שהוא מכיל רק את הכתובת של האיבר הראשון ברשימה



תבנית:

```
struct list
{
    data
    ...
    ...
    struct list *next;
};
```

כל איבר ברשימה הוא בעל המבנה הנ"ל כלומר מכיל בחלקו הראשון נתונים ובחלקו השני מצביע לאיבר הבא.
דוגמאות:

```
struct list
{
    int num;
    struct list *next;
};
```

```
struct student
{
    char f_name[10];
    char l_name[10];
    int frade;
    struct student *next;
};
```

נתבונן ב list (שהוגדר).
נניח כי list כ נגדיר את ראש הרשימה, (כמו בתרשים) 4 איברים בעלת head -.
מכיל את הכתובת של האיבר הראשון ברשימה. על מנת לפנות לאיבר הראשון נשתמש Head
שהוא מבנה עם שני שדות head כך נוכל לקבל את התוכן של הכתובת ב - *head : * באופרטור
המסומנים על ידי head->next ו head->num.

גם על מנת לגשת לשדות num ו- next ניתן בעזרת האופרטור *: (*head).next או (*head).num,
לכתיבה מקוצרת -> אך נשתמש באופרטור.

הוספת איברים לתחילת הרשימה (add)

```
void add (struct list** head, int num)
{
    struct list* second = *head;
    *head = (struct list*)malloc(sizeof(struct list));
    (*head)->num = num;
    (*head)->next = second;
}
```

הוא בעצם ראש הרשימה ומקובל לקרוא לו) יש משתנה מצביע לרשימה (main()נניח כי בפונקציה
שתוכנו הוא הכתובת של האיבר הראשון ברשימה. כשאנו מעוניינים לצרף איברים לרשימה head),
יש צורך לשלוח את הכתובת של ראש הרשימה (כדי לבצע עידכונים ברשימה) ולכן הפרמטר שנשלח
לפונקציה add הוא מצביע למצביע לרשימה - כך שהתוכן של המשתנה head החדש הוא הכתובת
של ראש הרשימה (head של הפונקציה main()). תוכנו של *head החדש הוא הכתובת של האיבר
הראשון שנוסף כעת לרשימה (הראשון הוא האיבר הראשון של הרשימה בכניסה הראשונה null כלומר
הרשימה ריקה).
בפונקציה add() אנו מצהירים על משתנה second מסוג רשימה שמטרתו לשמור על הכתובת של
האיבר הראשון ברשימה המקורית.
בשורה הבאה בונים את האיבר הראשון - מקצים כתובת חדשה וממלאים את השדות בערכים. לאחר
מכן האיבר החדש שלנו יצביע על האיבר הראשון הקודם (שהיה ריק) ולכן האיבר הראשון מצביע על
null.
בפעם הבאה נבנה שוב איבר ראשון חדש לרשימה אך הפעם הוא יצביע על האיבר הקודם של ראש
הרשימה והרי ערכו שונה מ- null. כך יבנו כל האיברים.

הצגת הרשימה (display)

```
void display_list (struct list* head)
{
    struct list* temp = head;
    printf ("\nThe members of the list are: \n\n");

    while (temp != NULL)
    {
        printf (" %d ", temp->num);
        temp = temp->next;
    }

    temp = head;
    printf("\n");
    while (temp != NULL)
    {
        printf ("%p ", temp);
        temp = temp->next;
    }
}
```

אסור לקדם את temp הוא הכתובת של האיבר הראשון ברשימה והוא נשמר במשתנה Head (מכיוון שאם נעשה זאת נאבד את המצביע ולא נוכל יותר לגשת למקומות אלו בזיכרון head.
מקבל את temp בתחילה מציגים את השדות של האיברים עד שמגיעים לסוף הרשימה אח"כ שוב הכתובת של האיבר הראשון ברשימה ואז מציגים את הכתובות של אברי הרשימה.

מחיקת הרשימה (del)

```
void del_member (struct list** head)
{
    struct list *temp = *head;
    *head = (*head)->next;
    printf ("%p\t", temp);
    free(temp);
}
```

```
void delete_list (struct list** head)
{
    printf("\n\nDeleting the list\n");
    while (*head)
```

```
        del_member (head);
}
```

את פעולת מחיקת הרשימה נעשה בשני שלבים - בשלב הראשון נעבור על איברי הרשימה כל עוד הרשימה לא ריקה ובשלב השני נמחק את האיברים. התהליך מתבצע בצורה הבאה:
הפונקציה del_member() מקבלת כארגומנט את הכתובת של ראש הרשימה (כעתhead החדש). כל עוד (את הכתובת של האיבר הראשון ברשימה *head -מכיל את הכתובת של ראש הרשימה ו שמכיל את הכתובת של ראש הרשימה לפונקציה head לא הגענו לסוף הרשימה היא שולחת את שם נשמרת הכתובת של האיבר הראשון של הרשימה במשתנה זמני הנקרא temp. del_member(), האיבר הראשון ברשימה מקודם, כלומר כעת האיבר הראשון ברשימה החדשה הוא האיבר את האיבר הראשון הישן וכך הלאה לכל האיברים. free השני ברשימה הקודמת, ומשחררים על ידי הפונקציה

כעת נראה את התוכנית במלואה:
דוגמא 1:

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

const int MAX = 3;

struct list
{
   int num;
   struct list *next;
};


void add (struct list** head, int num)
{
   struct list* second = *head;
   *head = (struct list*)malloc(sizeof(struct list));
   (*head)->num = num;
   (*head)->next = second;
}


void build_list (struct list** head)
{
   int i, num;
   for (i = 0; i < MAX; i++)
   {
      num = random(10);
      add(head, num);
```

```
    }
}


void display_list (struct list* head)
{
    struct list* temp = head;

    printf ("\nThe members of the list are: \n\n");

    while (temp != NULL)
    {
        printf (" %d ", temp->num);
        temp = temp->next;
    }

    temp = head;
    printf("\n");
    while (temp != NULL)
    {
        printf ("%p ", temp);
        temp = temp->next;
    }
}


void del_member (struct list** head)
{
    struct list *temp = *head;
    *head = (*head)->next;
        printf ("%p\t", temp);
    free(temp);
}

void delete_list (struct list** head)
{
    printf("\n\nDeleting the list\n");
    while (*head)
        del_member (head);
}


void main()
{

    struct list* head = NULL;
```

```
    randomize();

    build_list (&head);
    display_list (head);
    delete_list (&head);
}
```

הוספת איברים לסוף הרשימה (append)

```
void append (int num, struct number** head, struct number** last)
{
    struct number *newp =(struct number*)malloc(sizeof(struct number));
    newp->num = num; //newp = new pointer
    newp->next = NULL;

    if (*head == NULL)
        *head = newp;
    else
        (*last)->next = newp;
   *last = newp
}
```

כדי להוסיף איברים לסוף הרשימה יש צורך לשמור את הכתובת של סוף הרשימה. כתובת זו נשמרת במשתנה last ו-*head מכיל את הכתובת של האיבר הראשון ברשימה.
הוא זה שיהיה האיבר החדש והאחרון ברשימה (בכל יצירת איבר חדש) ולכן הוא יצביע על newp (סוף הרשימה). בודקים אם הרשימה ריקה ואין בה איברים (יקרה בכל רשימה חדשה שנבנה) null הוא האיבר החדש שבנינו, ואם לא כלומר הרשימה לא ריקה אז *head אז האיבר הראשון ברשימה היא הכתובת של האיבר האחרון ברשימה - נדאג *last.מצרפים את האיבר החדש לסוף הרשימה שיצביע על האיבר החדש מאחר last לעדכן את המצביע שלו לאיבר החדש שבנינו ואח"כ לעדכן את והוא החליף את האחרון הקודם. ניתן לראות כי האיבר האחרון מתעדכן כל פעם בהתאם לאיבר האחרון שנוסף.

דוגמא 2:

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

#define MAX 3

struct number
{
```

```
    int num;
    struct number *next;
};

void append (int num, struct number** head, struct number** last)
{
    struct number *newp =(struct number*)malloc(sizeof(struct number));
    newp->num = num; //newp = new pointer
    newp->next = NULL;

    if (*head == NULL)
        *head = newp;
    else
       (*last)->next = newp;
    *last = newp;
}

void read_2_list (struct number** head, struct number** last)
{
    int array[MAX];
    int i;
    for (i = 0; i < MAX; i++)
    {
       array[i] = i+random(20);
       append (array[i], head, last);
    }
}

void display_list (struct number* head, char *msg)
{
    struct number* temp = head;

    printf ("\n %s",msg);
    printf("\nThe address of the list is: %p", &head);
    printf ("\nThe members of the list are: \n\n");

    while (temp != NULL)
    {
       printf (" %d ", temp->num);
       temp = temp->next;
    }

    temp = head;
    printf("\n");
    while (temp != NULL)
    {
```

```
        printf ("%p ", temp);
        temp = temp->next;
    }
}

void swap (int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void sorting_list (struct number** head)
{
    int i, j;
    struct number* temp1 = *head;
    struct number* temp2;

    for ( ; temp1->next; temp1 = temp1->next)
    {
        temp2 = temp1->next;
        for ( ; temp2; temp2 = temp2->next)
        {
            if (temp2->num > temp1->num)
            swap(&temp2->num, &temp1->num);
        }
    }
}

void sum (struct number* head)
{
    int sum = 0;
    struct number* temp = head;

    for (; temp; temp = temp->next)
        sum += temp->num;

    printf ("\nThe sum of the numbers in the list = %d", sum);
    printf ("\nThe average of the numbers in the list = %f", (float)sum/MAX);
    printf ("\nThe max number in the list = %d", head->num);
}

void del_member (struct number** head)
{
    struct number *temp = *head;
    *head = (*head)->next;
```

```
    printf("%p\t",temp);
    free(temp);
}

void delete_list (struct number** head)
{
    printf("\n\nDeleting the list\n");
    while (*head)
    del_member (head);
}

void main()
{

    struct number* head = NULL;
    struct number* last;


    randomize();

    read_2_list (&head, &last);
    display_list (head, "Display list of number before sorting");
    sorting_list (&head);
    display_list (head, "Display list of number after sorting");
    sum (head);
    delete_list (&head);
}
```

הערה: למתעניינים במיונים - ישנם ספרים רבים בנושא, פה לא נתעמק בדרכי פעולתם.

דוגמא 3:

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

struct employee
{
    char f_name[10];
    char l_name[10];
    float wage;
};

struct node
{
```

```
    employee emp;
    struct node* next;
};

const MAX = 3;

employee read_data()
{
    employee p;

    printf("Enter your first name: ");
    flushall();
    gets(p.f_name);
    printf("Enter your last name: ");
    flushall();
    gets(p.l_name);
    printf("Enter your wage: ");
    flushall();
    scanf ("%f",&p.wage);
    return p;
}

void append (struct node* &head, employee emp)
{
    struct node* n_last = new node; //n_last = new last
    static struct node* last;
    n_last->emp = emp;
    n_last->next = NULL;

    if (head == NULL)
        head = n_last;
    else
        last->next = n_last;
    last = n_last;
}

void bulid_list (struct node* &head)
{
    int i;
    for (i = 0; i < MAX; i++)
        append (head, read_data());
}


void read_2_file (struct node* &head, FILE *f)
{
    f = fopen ("employee.txt", "wt");
```

```
    struct node* temp = head;

    while (temp != NULL)
    {
        fprintf (f, "%s %s %8.2f\n", temp->emp.f_name, temp->emp.l_name,temp-
>emp.wage);
        temp = temp->next;
    }
    fclose(f);
}


void del_node (struct node* &head)
{
    struct node* temp = head;
    head = head->next;
    free (temp);
}

void delete_list (struct node* &head)
{
    printf("\n\nDeleting the list");
    while (head)
    del_node (head);
}


void display_list (struct node* head)
{
    int i;
    struct node* temp = head;

    for (i = 0; temp ; i++)
    {
        printf ("\n %d", i);
        printf (" %s %s %8.2f ", temp->emp.f_name,
        temp->emp.l_name, temp->emp.wage);
        temp = temp->next;
    }
}

void sum_high (struct node* head)
{
    struct node* max = head;
    float sum = 0;
```

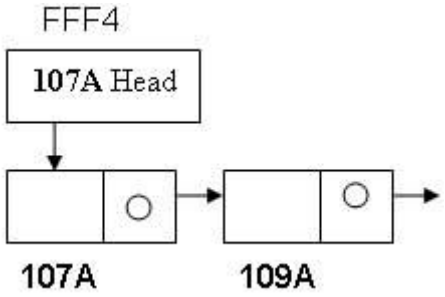
```
    while (head)
    {
        if (max->emp.wage < head->emp.wage)
        max = head;
        sum += head->emp.wage;
        head = head->next;
    }
    printf ("\n\nThe highest selary is: %6.2f", max->emp.wage);
    printf ("\nThe sum of the selaries is: %10.2f", sum);
    printf ("\n\nThe average of the selaries is: %6.2f",sum/MAX);
}

void main()
{
    randomize();
    struct node* head = NULL;
    FILE *f;

    bulid_list (head);
    read_2_file (head, f);
    display_list (head);
    sum_high (head);
    delete_list (head);
}
```

נסביר בהרחבה:

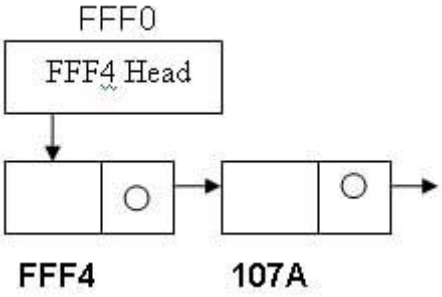
ואנו מעוניינים שהאיברים (fff4 נניח יושב בכתובת) הוא מטיפוס מצביע לרשימה head -מכיוון ש (כתובתו של head את כתובתו של build_list אז נשלח לפונקציה head מטיפוס רשימה יתווספו ל (aנניח בכתובת 107) מצויה הכתובת של האיבר הראשון ברשימה head ראש הרשימה). בתוך ובכתובת זו יש ערכים - שם ,שם משפחה , משכורת וכתובת של האיבר הבא. הפרמטר של build_list() בפונקציה head הוא מצביע למצביע כך שכעת ערכו של build_list() הפונקציה ישב (כתובת זה תמחק כאשר נצא מהפונקציה כי לא הקצנו אותה באופן fff0 בכתובת חדשה - נניח מסומן על ידי) fff4 המקורי, כלומר head-דינאמי) והערך היושב בכתובת זו יהיה הכתובת של ה build_list() של הפונקציה **head כלומר) וערכה של כתובת זו (build_list() של הפונקציה *head הוא האיבר הראשון של הרשימה המקורית.

אנו append. המקורי לפונקציה head כאשר אנו מוסיפים איברים אנו שולחים את הכתובת של הוא משתנה סטאטי כלומר ערכו מאותחל last כאשר n_nast -ו last - מצהירים על שתי משתנים בפעם הראשונה שנכנסים לפונקציה ולאחר מכן ערכו נשמר (בניגוד למשתנה רגיל המאותחל כל פעם שנכנסים לפונקציה וערכו מהכניסה הקודמת לפונקציה לא נשמר). תפקידו של משתנה זה מכיוון שכל פעם אנו מצרפים איבר לסוף אז משתנה זה) לשמור על האיבר האחרון של הרשימה הוא מטיפוס רשימה ולכן שדהו הראשון הוא n_last המשתנה .(append מעודכן כל כניסה לפונקציה שדהו השני הוא read_data.() שנשלח מהפונקציה emp ואנו מזינים את השדה הזה ב employee יהיה כל פעם האיבר האחרון n_last מצביע לאיבר מטיפוס רשימה אך מכיוון שאנו מעוניינים ש *head (כלומר סוף הרשימה). בודקים אם NULLאז רושמים כי שדהו השני יהיה (new last) החדש אז זה סימן NULL מכיל את הערך (main המקורי של פונקצית head שהוא הכתובת של ה) שהרשימה ריקה ולכן סוף הרשימה שווה לראש הרשימה. אחרת (כלומר הרשימה לא ריקה)

והשורה לאחריו תמיד תתבצע (בין אם הרשימה ריקה או לא) - n_last יכיל את last המשתנה
(כלומר סוף הרשימה מתעדכנת) n_last מקבל את כתובתו של last המשתנה.
את כתובתו של ראש הרשימה (append() כמו הפונקציה) מקבלת read_2_list() הפונקציה
ומדפיסה את ערכו של temp מאחסנת ערך זה במשתנה, (של הפונקציה head* המאוחסן ב)
מכיוון שאם לא (temp) חשוב להשתמש במשתנה זמני . "employee.txt" לקובץ temp המשתנה
ואז הינו מאבדים את האפשרות לגשת לכתובות הקודמות head* הינו עושים זאת הינו מקדמים את
אין זה משפיע על המצביע של ראש הרשימה temp ולבדוק את ערכן. כשמקדמים את.
head* הוא מצביע למצביע - אזי כמו שהסברנו קודם ערכו של Del_list() הפרמטר של הפונקציה
(temp), יהיה הכתובת של ראש הרשימה המקורי, ולכן אנו שומרים כתובת זו במשתנה זמני
ומשחררים את הכתובת של ראש הרשימה head = *head->next* מקדמים את ראש הרשימה
הקודם על ידי פקודת free.
אין צורך לשלוח את כתובתו של ראש הרשימה (כי לא מצרפים או מוחקים display() בפונקציה
display() איברים - בסך הכל רוצים להציג את הערכים של הרשימה) ולכן הפרמטר של הפונקציה
הוא מצביע המכיל את התוכן של ראש הרשימה (ואילו תוכנו של ראש הרשימה הוא הכתובת של
האיבר הראשון ברשימה) וכך שומרים את הכתובת של האיבר הראשון ברשימה מדפיסים את ערכיו
ומתקדמים לאיבר הבא.



לפונקציה שהפרמטר שלה הוא מצביע למצביע (&head) head כאשר אנו שולחים את כתובתו של
התמונה נראית כך:



 בניית רשימה על ידי רקורסיה

```
void rec_append (struct node** head, int i)
{
   if(*head)
   rec_append (&(*head)->next,i);
```

```
    else
    {
        *head = (struct node*)malloc(sizeof(struct node));
        (*head)->num = i;
        (*head)->next = NULL;
    }
}
```

את הכתובת של האיבר הראשון ברשימה. *head, מכיל את הכתובת של ראש הרשימה head
ולכן מקצים באופן דינאמי מקום בזיכרון שווה-null ל *head מכיוון שבתחילה הרשימה ריקה אז
(null כאשר השדה השני הוא) (האיבר הראשון ברשימה) וקובעים את שדותיו *head ל.
(*head)-> ולכן נקרא לפונקציה כאשר null -יהיה שונה מ *head בפעם הבאה שנכנס לפונקציה
הוא האיבר השני ברשימה. מכיוון שזו הפעם השנייה שאנו נכנסים לרשימה - עוד אין איבר next>
אז אנו נשלח את כתובתה לקריאה הבאה לפונקציה. כעת בכניסה הזו, null, שני, הערך יהיה
ולכן מקצים ל"ראש הרשימה החדש" (שהוא האיבר השני) מקום בזיכרון שווה-null ל*head
כעת יוצאים. (null מכיוון שהוא יהיה האיבר האחרון אז שדה השני יהיה) ומאתחלים את שדותיו
מהעותק השני של הפונקציה לעותק הראשון וניתן לראות כי נוסף לנו האיבר שיצרנו לרשימה וכך
הלאה - כלומר בונים את האיבר השלישי על ידי כניסה 3 פעמים לפונקציה, קובעים ערכים לאיבר
יוצאים מהעותקים עד לעותק הראשון וכך הלאה.

מחיקת איברים על ידי רקורסיה

```
void del_list (struct node** head)
{
    if(*head)
    {
        printf("%p ",(*head)->next);
        del_list (&(*head)->next);
        free(*head);
    }
    else
    {
        gotoxy (1, wherey()-1);
        printf ("Destruct list by deleting adresses: \n\n");
    }
}
```

את הכתובת של האיבר הראשון. בכניסה *head, מכיל את הכתובת של ראש הרשימה head
הוא לדוגמא 15 . אזי מדפיסים את השני ברשימה y-וערך ה null-שונה מ *head הראשונה
וקוראים לפונקציה עם הכתובת של האיבר השני ברשימה. בכניסה הזו לפונקציה (*head)->next))
הוא האיבר השני ברשימה וכך הלא עד שנכנסים לקריאה האחרונה (האיבר האחרון). *head
מדפיסים את המשפט ויוצאים מהעותק האחרון. else- נכנסים ל null, שווה ל *head במקרה זה
(ולכן לא נדפיס את המשפט שוב) if- כזכור בעותק הלפני אחרון נכנסנו לגוף של פקודת ה.
הוא האיבר האחרון - אנו חוזרים לשורה שלאחר הקריאה לפונקציה ולכן נותר רק לבצע *head
שחרור של האיבר האחרון ולחזור לקריאה הקודמת בה משוחרר האיבר לפני אחרון וכך הלאה עד
לאיבר הראשון.

כעת נאחד את הפונקציות שהראנו לתוכנית אחת:
בניית צמתים על ידי רקורסיה, מחיקת צמתים על ידי רקורסיה.
דוגמא 4:

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

const int MAX = 5;

struct node
{
    int num;
    struct node *next;
};

void rec_append (struct node** head, int i)
{
    if(*head)
        rec_append (&(*head)->next,i);
    else
    {
        *head = (struct node*)malloc(sizeof(struct node));
        (*head)->num = i;
        (*head)->next = NULL;
    }
}

void bulid_list (struct node** head)
{
    int i;
    for(i = 0; i < MAX; i++)
        rec_append (head, i+1);
}

void display_list (struct node* head)
{
    printf ("\n %s");
    printf ("\nThe members of the list are: \n\n");

    while (head != NULL)
    {
        printf (" %d ", head->num);
        printf (" %p \n", head->next);
        head = head->next;
    }
}

void del_list (struct node** head)
```

```
{
    if(*head)
    {
        printf("%p ",(*head)->next);
        del_list (&(*head)->next);
        free(*head);
    }
    else
    {
        gotoxy (1, wherey()-1);
        printf ("Destruct list by deleting adresses: \n\n");
    }
}

void delete_list (struct node** head)
{
    printf("\n\nDeleting the list");
    while (*head)
    del_list (head);
}

void main()
{
    struct node* head = NULL;

    bulid_list (&head);
    display_list (head);
    del_list (&head);
}
```

רשימות עם יותר ממצביע אחד

ציינו שרשימה היא מבנה נתונים שבחלקו האחד כולל נתונים ובחלקו האחר מכיל מצביע לאיבר הבא ברשימה. חלק זה יכול לכלול מספר מצביעים לדוגמא: מצביע לאיבר הקודם ומצביע לאיבר הבא או מצביעים לשני איברים עוקבים (עץ בינארי) או מצביעים לשדות שונים.

יצירת רשימה עם שני מצביעים:

```
void create_list (struct student** head)
{
    struct student* b_last; //b_last = before last
    struct student* last;
    int i;
    leng = MAX/2 + random(MAX/2);

    printf ("\n\nInput %d random id numbers: \n");
    b_last = *head;
```

```
    for (i = 0; i < leng; i++)
    {
        last = (struct student*)malloc(sizeof(struct student));
        last->id = random(1000)+6000;
        last->grade = random(51)+50;
        last->next_id = NULL;
        last->next_grade = NULL;
        b_last->next_id = last;
        b_last->next_grade = last;
        b_last = last;
    }
}
```

המייצגים את האיבר הלפני אחרון והאחרון שברשימה. b_last, last - מצהירים על שתי משתנים
מכיל את הכתובת של האיבר הראשון של הרשימה, כעת נראה איך (b_last) האיבר הלפני אחרון
בונים את הרשימה על ידי שימוש בלולאה:
מאתחלים את שדותיו, (last) אנו מקצים מקום לאיבר חדש שהוא יהיה האיבר האחרון ברשימה
יהיה האיבר האחרון(. לאחר last-כי אנו מעוניינים ש) null -המספריים, ואת שדותיו הכתובתיים ל
שהוא כזכור האיבר הראשון של הרשימה (מצרפים את אברי last להצביע על b_last מכן גורמים ל
ושוב באיטרציה הבאה מקצים last מכיל את b_last הרשימה החדשים לסוף הרשימה) ולבסוף
מקוmlast ל -b_last וגורמים ל last להצביע על ש וכך הלאה עד i=leng .
הן אותן כתובות id-והמצביע ל grade -יש לשים לב כי הכתובות של המצביע ל.

id. כעת נציג את התוכנית הבונה רשימה עם שני מצביעים וכן ממיינת את הרשימה לפי השדה
דוגמא 5:

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

#define MAX 7

int leng;

struct student
{
    long id;
    int grade;
    struct student* next_id;
    struct student* next_grade;
};

void create_list (struct student** head)
{
    struct student* b_last;
```

```
    struct student* last;
    int i;
    leng = MAX/2 + random(MAX/2);

    printf ("\n\nInput %d random id numbers: \n");
    b_last = *head; //b_last = before last

    for (i = 0; i < leng; i++)
    {
      last = (struct student*)malloc(sizeof(struct student));
      last->id = random(1000)+6000;
      last->grade = random(51)+50;
      last->next_id = NULL;
      last->next_grade = NULL;
      b_last->next_id = last;
      b_last->next_grade = last;
      b_last = last;
    }
}

void write_list (struct student* head)
{
    struct student* temp_id = head->next_id;
    struct student* temp_grade = head->next_grade;

    printf ("\nLIst by id numbers: \n");
    while(temp_id)
    {
      printf ("\nId: %ld\t%p", temp_id->id, temp_id->next_id);
      temp_id = temp_id->next_id;
    }
    printf ("\nLIst by grades: \n");
    while(temp_grade)
    {
      printf ("\nGrade: %d\t%p", temp_grade->grade, temp_grade->next_grade);
      temp_grade = temp_grade->next_grade;
    }
}

struct student* place (struct student* head, int indx)
{
    struct student* temp = head;
    int i;
    for (i = 0; i < indx; i++)
      temp = temp->next_id;
    return temp;
}

int place_id (struct student* head, int x)
```

```
{
    int i;
    struct student* temp;
    temp = head;

    for (i = 0; i < x; i++)
       temp = temp->next_id;
    return temp->id;
}

int bin_search (struct student* head, int leng, long x)
{
    int first = 1;
    int last = leng;
    int mid = (first+last)/2;

    while (first < last)
    {
       if (x < place_id (head, mid))
          last = mid;
       else
          first = mid+1;
       mid = (first+last)/2;
    }
    if (x == place_id (head, mid))
       return mid;
    else
       return first;
}

void insert (struct student* head, int leng, long x)
{
    struct student* temp;
    struct student* temp1;
    int new_indx;

    new_indx = bin_search (head, leng, x);
    if (new_indx == leng) return;
    temp = place (head, new_indx); //new position in the list
    temp1 = place (head, leng); //points to the last
    place (head, leng-1)->next_id = temp1->next_id;
    place (head, new_indx-1)->next_id = temp1;
    temp1->next_id = temp;
}

void binary_sort (struct student** head)
{
    struct student* temp;
    int i;
```

```
    printf ("\n\nThe list after sorting by id numbers:\n\n");
    temp = *head;
    for (i = 0; i <= leng; i++)
    {
      temp = place (*head, i);
      insert (*head, i, temp->id);
    }
}

void del_member (struct student** head)
{
    struct student *temp = *head;
    *head = (*head)->next_id;
    printf ("%p\t", temp);
    free(temp);

}

void del_list (struct student** head)
{
    printf("\n\nDeleting the list\n");
    while (*head)
      del_member (head);
}

void main()
{
    int leng;
    struct student* head;
    randomize();
    head = (struct student*)malloc(sizeof(struct student));
    head->id = 0;
    head->grade = 0;

    create_list (&head);
    write_list (head);
    binary_sort(&head);
    write_list (head);
    del_list (&head);
}
```

רשימה של פרטי סטודנט הכוללים ת.ז וציון, מצביע לציון ומצביע לת.ז. (למעשה אלו מצביעים לטיפוס 'סטודנט').

מיון הרשימה על פי ת.ז. - המיון שהוצג הוא מיון בינארי (למעוניין לקרוא על מיונים ימצא זאת בספרות על מבנה נתונים כפי שהזכרנו, לא נפרט את אופיו של המיון).

כדי לבצע מיון אנו יוצרים ראש דמה והמיון מתבצע ממנו והלאה (הנתונים בו לא רלוונטיים) - משתמשים בו כדי לא לאבד כתובות ברשימה לכן במחיקת הכתובות נוספו 2 כתובות, אחת מהן היא

בס"ד

בעבור ראש הדמה והשנייה בעבור המצביע לראש הדמה.
מכיוון שהמצביע לת.ז ולציון בעלי אותה כתובת יש למחוק פעם אחת כל כתובת (כלומר למחוק מצביע לציון או מצביע לת.ז).
בהצגת הרשימה הודפסו הנתונים (ת.ז או ציון) בטור אחד ובשני השדה השני של הרשימה כלומר הכתובת של האיבר הבא</br>.

מחסנית ותור

פרק זה דן במחסניות ותורים שהם למעשה ואריאציה על מבני נתונים של רשימה. מכיוון שדנו בנושא בפרק שעבר בהרחבה נתמקד כאן יותר בתכונות הייחודיות למחסנית ותור.

כלומר - (Abstract Data Type) המחסנית והתור שייכים לקבוצה הנקראת מבנה נתונים מופשט הם מאופיינים ע"י תכונות ייחודיות ועל ידי קבוצת פעולות בסיסיות.

המחסנית (stack)

מחסנית - הינה מבנה נתונים מופשט. התכונה העיקרית שלה היא שהאיבר הראשון שיוכנס אליה כלומר הוצאה והכנסה של איבר אפשרית רק מלמעלה - (first in last out) יהיה הראשון לצאת (מראש המחסנית).

Last
..
Fourth
Third
Second
First
המחסנית

ראש המחסנית הוא האיבר העליון.

הפעולות:
push - דחיפת איבר לראש המחסנית.
pop - (שליפת ראש המחסנית) שליפת איבר מראש המחסנית.
init - אתחול המחסנית,האיבר הראשון של המחסנית שווה ל null

נממש את המחסנית באמצעות רשימה שבה כל פעולות ההכנסה וההוצאה מתבצעות בקצה אחד בלבד.

המחסנית הינה רשימה שהפעולות של הכנסת איברים והוצאתם נעשית בכיוון אחד ולכן מבנהו של המחסנית זהה לשל הרשימה (רק כאן אנו נגדיר ששמו של המבנה הוא stack ולא list על מנת שהדבר יהיה ברור).

מבנה של המחסנית:

```
struct stack
{
    ...
    struct stack *next;
};
```

הכנסת איבר לראש המחסנית - push():

```
void push (struct stack** head, int num)
{
    struct stack* second = *head;
    *head = (struct stack*)malloc(sizeof(struct stack));
    (*head)->num = num;
    (*head)->next = second;
}
```

stack (מחסנית) הוא מבנה רשימתי כלומר מכיל שדות רגילים בחלקו הראשון וחלקו השני מצביע לאיבר הבא.
הוא מצביע למצביע (המכיל את הכתובת של ראש) הפרמטר של הפונקציה push() (כשמו כן הוא - יכיל את second המחסנית/רשימה). האיבר הראשון של המחסנית נשמר במשתנה האיבר השני של המחסנית), מקצים מקום בזיכרון לראש החדש של המחסנית, מאתחלים את שדהו הפשוט (אם יש מספר כאלה אז דואגים להקנות להם ערכים) ואת שדהו המצביע שיצביע על הראש הקודם של המחסנית (כזכור ערך זה נשמר במשתנה second).
מקצים null (=second). שווה ל *head בפעם הראשונה שנכנס לפונקציה, המחסנית ריקה ולכן מקום בזיכרון לאיבר הראשון של המחסנית, מאתחלים את שדהו הפשוט ואחר כך את שדה המצביע שיצביע ל- null (senond = null כי). בפעם הבאה ניצור שוב איבר ראשון חדש למחסנית והוא יצביע לאיבר הראשון שכעת יצרנו.

הוצאת איבר מראש המחסנית - pop():

```
void pop (struct stack** head)
{
    if (*head == NULL)
    {
        printf ("\n can't pop from empty stack\n");
        return;
    }
    else
    {
        struct stack *temp = *head;
        *head = (*head)->next;
        free(temp);
```

```
    }
}
```

הוא מצביע למצביע (המכיל את הכתובת של ראש push() הפרמטר של הפונקציה
מכיל את הכתובת של האיבר הראשון *head) המחסנית/רשימה). בודקים אם יש איברים במחסנית
ומסיימים אם המחסנית ריקה. אחרת יש צורך למחוק את האיבר שבראש המחסנית. (במחסנית
המחיקה מתבצעת באופן הבא:
שומר את הכתובת של האיבר הראשון במחסנית. מקדמים את האיברים במחסנית temp המשתנה
כלומר האיבר הראשון החדש במחסנית יהיה האיבר השני במחסנית הקודמת (לפני שקידמנו איבר)
ולבסוף משחררים את הזיכרון של האיבר הראשון הקודם.

כעת נממש את שתי הפונקציות שהוסברו בתוכנית אחת:
דוגמא 1:

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

const int MAX = 3;

struct stack
{
    int num;
    struct stack *next;
};

void init_stack (struct stack** head)
{
    *head = NULL;
}


void push (struct stack** head, int num)
{
    struct stack* second = *head;
    *head = (struct stack*)malloc(sizeof(struct stack));
    (*head)->num = num;
    (*head)->next = second;
}


void build_stack (struct stack** head)
{
    int i, num;
    printf ("\nThe number by their insertion order are:\n");
```

```
    for (i = 0; i < MAX; i++)
    {
        num = random(10);
        printf ("%d\t", num);
        push (head, num);
    }
}


void display_stack (struct stack* head)
{
    struct stack* temp = head;

    printf ("\nThe members of the stack are: \n\n");

    while (temp != NULL)
    {
        printf (" %d ", temp->num);
        temp = temp->next;
    }

    temp = head;
    printf("\n");
    while (temp != NULL)
    {
        printf ("%p ", temp);
        temp = temp->next;
    }
}


void pop (struct stack** head)
{
    if (*head == NULL)
    {
        printf ("\n can't pop from empty stack\n");
        return;
    }
    else
    {
        struct stack *temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}
```

```
void delete_stack (struct stack** head)
{
    printf("\n\nDeleting the stack:\n");
    while (*head)
    {
        printf ("%p\t", *head);
        pop (head);
    }
}

void main()
{
    struct stack* head;
    randomize();

    init_stack (&head);
    build_stack (&head);
    display_stack (head);
    delete_stack (&head);
}
```

העבר את העכבר מעל הדוגמא כדי לראות הסבר מפורט

תור

מבנה נתונים מופשט. תכונתו העיקרית היא הכנסה והוצאה של איברים דרך התאים שבקצוות - הכנסה של האיברים דרך קצה אחד והוצאתם דרך הקצה האחר.

⇦	Last	...	Third	Second	First	⇦

הכנסה מהתא הימני ביותר, הוצאה מהתא השמאלי ביותר.
התא השמאלי ביותר הוא ראש התור.
האיבר הראשון שהוכנס לתור הוא יהיה הראשון לצאת (first in first out).

הפעולות:
append - הוספת איבר לתור.
Remove - סילוק איבר מהתור.
init - אתחול התור.

נממש את התור באמצעות רשימה שבה כל פעולות ההכנסה מתבצעת דרך קצה אחד ופעולת

ההוצאה מתבצעות דרך הקצה האחר.

דוגמא 2:

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

const int MAX = 5;

struct queue
{
    int num;
    struct queue *next;
};

void init_queue (struct queue** que)
{
    *que = NULL;
}

void append (struct queue** que, int num)
{
    static struct queue* last;
    static struct queue* n_last;

    n_last = (struct queue*)malloc(sizeof(struct queue));
    n_last->num = num;
    n_last->next =NULL;

    if (*que == NULL)
        *que = n_last;
    else
        last->next = n_last;
    last = n_last;
}

void build_queue (struct queue** que)
{
    int i, num;

    printf ("\nThe number by their insertion order are:\n");
    for (i = 0; i < MAX; i++)
    {
        num = random(10);
        append (que, num);
    }
}
```

```
void display_queue (struct queue* que)
{
    struct queue* temp = que;

    printf ("\nThe members of the queue are: \n\n");

    while (temp != NULL)
    {
        printf (" %d ", temp->num);
        temp = temp->next;
    }

    temp = que;
    printf("\n");
    while (temp != NULL)
    {
        printf ("%p ", temp);
        temp = temp->next;
    }
}

void delete_queue (struct queue** que)
{
    struct queue *temp;

    printf("\n\nDeleting the queue:\n");
    while (*que)
    {
        printf ("%p\t", *que);
        temp = *que;
        *que = (*que)->next;
        free(temp);

    }
}

void main()
{
    struct queue* que;
    randomize();

    init_queue (&que);
    build_queue (&que);
    display_queue (que);
    delete_queue (&que);
}
```

בס"ד

עצים בינאריים

עץ בינארי - מבנה נתונים מופשט הבנוי הצורה הפוכה לעץ טבעי כלומר שורשו למעלה והסתעפותו כלפי מטה, כך שבתחתית מצויים ענפיו ועליו.
הצומת העליון הוא שורש העץ. כל נקודת הסתעפות נקראת צומת ולכל צומת שני ענפים, שמאלי וימני (במקרה של עץ בינארי). שני צמתים של ענפים אלו נקראים בנים או ילדים (ימני ושמאלי) של צומת ההסתעפות וצומת ההסתעפות נקרא אב של שני בנים אלו.

הערה: העץ נקרא בינארי מכיוון שלכל צומת יש לכל היותר שני בנים. ניתן לייצר גם עצים תרינאריים ויותר על ידי הגדלת מספר הבנים לכל צומת.
כל צומת מהווה גם היא שורש של תת עץ (עץ נתונים הוא בעל דמיון עצמי מושלם).
לכל צומת יש הורה אחד בלבד.
עלה הוא צומת שאין לו ילדים, צומת פנימי הוא צומת שאינו עלה.
לצאצא (במסלול כל שני צמתים עוקבים מקיימים x אם קיים מסלול מצומת x צומת יקרא צאצא של יחס אב-בן).
שני צמתים יקראו אחים אם יש להם את אותו הורה.
גובה של צומת - אורך המסלול (מספר צמתים) מהצומת ועד לעלה.
גובה עץ - גובה של השורש.
גודל העץ - מספר צמתיו.
יתכן כי לצומת יהיה בן אחד בלבד.

צומת מכילה נתונים ושני מצביעים: אחד לבן ימני והשני לבן שמאלי.

מאחר וכל צומת הוא שורש של תת עץ - כל הפעולות על עצים הן רקורסיביות.
ערכו של בן ימני יהיה גדול מערכו של צומת האב וערכו של צומת שמאלי קטן מערכו של צומת האב.
לכן בהכנסת הנתונים נבדוק את הערך של הנתון שאנו מעוניינים להכניס לעץ - אם ערכו קטן מהשורש נפנה לבן השמאלי של השורש ואם לא נפנה לבן הימני של השורש.
כשנפעיל פעולה כזו ברקורסיה נתייחס כעת לצומת החדש כשורש של תת העץ שהתקבל. נשווה את הערך שאנו מבקשים להכניס לעץ עם השורש - אם קטן נפנה לבנו השמאלי ואם לא נפנה לבנו הימיני וכך הלאה עד אשר נמצא את המיקם המתאים לערך שלנו.

פעולות בסיסיות

init - אתחול העץ.
add, set - הכנסה לעץ.
del - מחיקת העץ.

תבנית של עץ:

```
struct tree
{
    type1 data1;
    type2 data2;
    ....
    struct tree* left;
```

```
    struct tree* right;
};
```

כמו ברשימה יש מצביע לעץ שמכיל את הכתובת של שורש העץ, כל צומת בעץ מכיל את השדות של המבנה tree ושני מצביעים לבנים.

add() הכנסת איבר לעץ

```
void add (struct tree** t, int x)
{
    if (*t == NULL)
        set (t, x);
    else if (x < (*t)->num)
        add (&(*t)->left, x);
    else
        add (&(*t)->right, x);
}
```

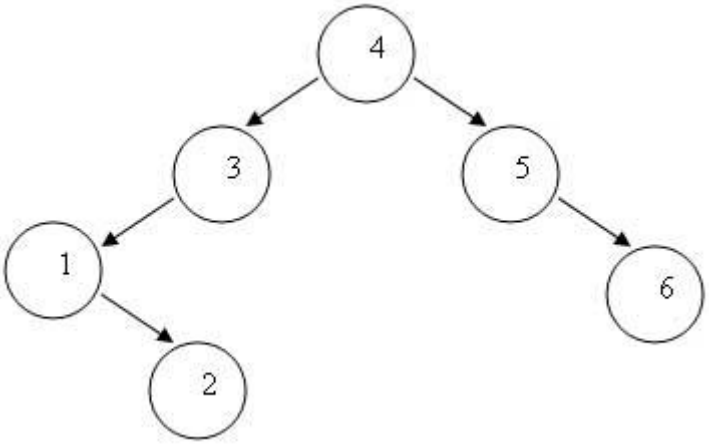
הפרמטרים שלה הם הערך שרוצים add() הכנסת האיבר מתבצעת על ידי פונקציה רקורסיבית. אם יש כמה כאלו יש לשלוח את) שהצומת (שמתפקדת בעץ כמו איברים שראינו ברשימות) תקבל הנתונים כפרמטרים לפונקציה add() והפרמטר הנוסף - מצביע למצביע שמכיל את הכתובת של (שורש העץ.
בהכנסת איבר חדש יש להתאים את מקומו על פי הערך שרוצים להכניס לעץ (על פי התבנית שכל תת שורש מכיל בן שמאלי וימני כאשר הבן השמאלי מכיל ערך קטן מהשורש והבן הימני מכיל ערך גדול או שווה לשורש). כשמגיעים למקום המתאים, בונים צומת לפי הפונקציה set: הפונקציה set() מקבלת כתובת וערך, היא מקצה זיכרון לצומת החדש, מציבה את הערך המבוקש ומאתחלת את שני המצביעים של הצומת החדש ל- null - בעצם הפונקציה set כלומר הצומת החדש הוא עלה, בונה עלים לעץ.

display() הצגת אברי העץ

```
void display_t (struct tree* t)
{
    if (t != NULL)
    {
        display_t (t->left);
        printf ("%d\t", t->num);
        display_t (t->right);
    }
    else
        return;
}
```

גם הצגת איברים מתבצעת על ידי פונקציה רקורסיבית הפועלת כך:
לפונקציה נשלח האיבר הראשון של העץ. אם העץ לא ריק אז מציגים את תת עץ שמאלי - ניתן עם השורש של התת עץ השמאלי. לאחר display() לראות זאת מכיוון שיש קריאה לפונקציה עולים לאב של העלה ואז ,(num) שהגענו לעלה השמאלי ביותר מדפיסים את השדה הפשוט שלו הפעם עם התת עץ הימני של הצומת שעליה דיברנו וכך הלאה display() יש שוב קריאה לפונקציה עם תת העץ הימני display() עד שמגיעים לשורש העץ, מדפיסים את ערכו ושוב קוראים לפונקציה של השורש.

עוברת על צמתי העץ display() להלן תרשים של עץ, נראה איך הפונקציה:
המספרים בצמתים הם הערכים שהצמתים מכילים - ערך השדה הפשוט (במקרה זה - שדה מספרי) במבנה האיבר מסוג צומת של עץ (הם לא מציינים את סדר הטיפול בצמתים או סדר ההכנסה או כל משמעות אחרת).



הפונקציה מתחילה בשורש - 4.
הצומת לא ריק לכן קוראים לפונקציה עם הצומת 3.
צומת 3 לא ריק ולכן קוראים לפונקציה עם הצומת 1.
הבן השמאלי של צומת 1 הוא null) הצומת 1 לא ריק ולכן קוראים לפונקציה עם null).
הצומת ריק וחוזרים לקריאה הקודמת עם צומת 1, מדפיסים את ערכו וקוראים לפונקציה עם 2.
הבן השמאלי של צומת 1 הוא null) הצומת לא ריק ולכן קוראים לפונקציה עם null).
הצומת ריק וחוזרים לקריאה הקודמת עם צומת 2, מדפיסים את ערכו וקוראים לפונקציה עם null.
הצומת ריק וחוזרים לקריאה הקודמת עם צומת 2, חוזרים לקריאה הקודמת עם צומת 1, חוזרים בנו הימיני של צומת) null לקריאה הקודמת עם צומת 3, מדפיסים את ערכו וקוראים לפונקציה עם 3).
הצומת ריק וחוזרים לקריאה הקודמת עם צומת 3, חוזרים לקריאה הקודמת עם צומת 4, מדפיסים את ערכו, וקוראים לפונקציה עם צומת 5.
בנו השמאלי של 5) הצומת לא ריק ולכן קוראים לפונקציה עם null).
הצומת ריק ולכן חוזרים לקריאה הקודמת עם צומת 5, מדפיסים את ערכו וקוראים לפונקציה עם הצומת 6.
הבן השמאלי של צומת 6) הצומת לא ריק ולכן קוראים לפונקציה עם null).
הצומת ריק וחוזרים לקריאה הקודמת עם צומת 6, מדפיסים את ערכו וקוראים לפונקציה עם הצומת הבן הימיני של צומת 6) null).

הצומת ריק וחוזרים לקריאה הקודמת עם צומת 6, חוזרים לקריאה הקודמת עם צומת 5, חוזרים לקריאה הקודמת עם צומת 4, לעותק הראשון והפונקציה מסיימת.
נרשום רק את אשר הודפס למסך:
צומת 1, צומת 2, צומת 3, צומת 4, צומת 5, צומת 6 - יש לשים לב כי פונקצית הדפסה זו מדפיסה את האיברים לפי סדר מהקטן לגדול.

מחיקת העץ del_t()

```
void del_t (struct tree** t)
{
    if (*t != NULL)
    {
        del_t (&(*t)->left);
        del_t (&(*t)->right);
        printf ("%p\t",*t);
        free (*t);
    }
    else
        return;
}
```

הפונקציה מקבלת את הכתובת של המצביע לשורש העץ. אם שורש העץ לא ריק קוראים לפונקציה עם הכתובת של השורש של תת העץ השמאלי, אם השורש של תת העץ השמאלי לא ריק שוב קוראים לפונקציה עם הכתובת של השורש של תת העץ של תת העץ של השורש של העץ וכך הלאה ולכן משחררים אותו על ידי פקודת delete. שני בניו הם null עד שמגיעים לעלה השמאלי ביותר.
עולים בקריאות הרקורסיביות ושוב - אם תת העץ הימני לא ריק נכנסים לתת העץ הימני וכו' (התהליך של הקריאות הרקורסיביות דומה להצגת העץ בהבדל המזערי - רק לאחר שתי הקריאות הרקורסיביות מדפיסים את הכתובת ומשחררים את הזיכרון של הצומת). בסיום משחררים את שורש העץ.
נרשום בקצרה על פי העץ שלנו מהתרשים באיזה סדר משתחררים הצמתים:
צומת 2, צומת 1, צומת 3, צומת 6, צומת 5, צומת 4.
נאחד את כל הפונקציות לתוכנית אחת:

דוגמא 1:

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

const MAX = 7;

struct tree
{
```

```
    int num;
    struct tree* left;
    struct tree* right;
};

void init (struct tree** t)
{
    *t = NULL;
}

void set (struct tree** t, int x)
{
    *t = (struct tree*)malloc(sizeof(struct tree));
    (*t)->num = x;
    (*t)->right = NULL;
    (*t)->left = NULL;
}

void add (struct tree** t, int x)
{
    if (*t == NULL)
        set (t, x);
    else if (x < (*t)->num)
        add (&(*t)->left, x);
    else
        add (&(*t)->right, x);
}

void bulid_t (struct tree** t)
{
    int i, x;

    for (i = 0; i < MAX; i++)
    {
        x = random(30);
        add (t,x);
    }
}


void display_t (struct tree* t)
{
    if (t != NULL)
    {
        display_t (t->left);
        printf ("%d\t", t->num);
        display_t (t->right);
```

```
    }
    else
        return;
}


void del_t (struct tree** t)
{
    if (*t != NULL)
    {
        del_t (&(*t)->left);
        del_t (&(*t)->right);
        printf ("%p\t",*t);
        free (*t);
    }
    else
        return;
}


void main()
{
    struct tree* t;

    randomize();

    init (&t);
    bulid_t (&t);
    display_t (t);
    printf ("\nDeleting the tree in address %p :\n",&t);
    del_t (&t);
}
```

העבר את העכבר מעל הדוגמא כדי לראות הסבר מפורט

עץ מלא- עץ שלכל צומת בו יש שני בנים.
עץ שלם- עץ שבו לכל רמה i יש 2 צמתים.

דוגמא 2:

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

```
struct tree
{
    int num;
    struct tree* left;
    struct tree* right;
};

void init (struct tree** t)
{
    *t = NULL;
}

void set (struct tree** t, int x)
{
    *t = (struct tree*)malloc(sizeof(struct tree));
    (*t)->num = x;
    (*t)->right = NULL;
    (*t)->left = NULL;
}

void add (struct tree** t, int x)
{
    if (*t == NULL)
        set (t, x);
    else if (x < (*t)->num)
        add (&(*t)->left, x);
    else
        add (&(*t)->right, x);
}
void bulid_t (struct tree** t)
{
    int i, x, max;
    max = 1 + random(5);
    for (i = 0; i < max; i++)
    {
        x = random(30);
        add (t,x);
    }
}

void display_t (struct tree* t)
{
    if (t != NULL)
    {
        printf ("%d\t", t->num);
        display_t (t->left);
        display_t (t->right);
    }
```

```
    else
        return;
}

int full_bin_t (struct tree** t)
{
    if (((*t)->left == NULL)&& ((*t)->right == NULL))
        return 1;
    else if (((*t)->left == NULL)|| ((*t)->right == NULL))
        return 0;
    else if (full_bin_t (&(*t)->left) && full_bin_t (&(*t)->right))
        return 1;
    else
        return 0;
}

void del_t (struct tree** t)
{
    if (*t != NULL)
    {
        del_t (&(*t)->left);
        del_t (&(*t)->right);
        printf ("%p\t", *t);
        free (*t);
    }
    else
        return;
}

void main()
{
    struct tree* t;
    randomize();

    init (&t);
    bulid_t (&t);
    display_t (t);
    if (full_bin_t (&t))
        printf ("\nThe tree is a full binary tree\n");
    else
        printf ("\nThe tree is not a full binary tree\n");
    del_t (&t);
}
```

הערה:

כאשר אנו שולחים כארגומנט את העץ יש לשלוח עם אופרטור הכתובת על מנת שהשינויים של הוספה לעץ ומחיקתו ישמרו אך כפרמטר הפונקציה יש להשתמש במצביע למצביע כך שהמצביע מכיל משתנה מסוג מצביע (המצביע לעץ). כך לדוגמא אם המצביע לעץ מכיל את הכתובת 0a88 אז המשתנה מצביע למצביע יהיה בכתובת מסוימת המכילה את fff4 וכתובתו של מצביע זה היא הכתובת fff4 שתכיל את הכתובת של האיבר הראשון של העץ.
כלומר מחוץ לפונקציות הנקראות:

```
&t = fff4
t = 0a88
*t האיבר הראשון של העץ.
בקריאה לפונקציות:
t = fff4
*t = 0a88
**t האיבר הראשון של העץ.
```

מבחן 1. שורש העץ מכיל: (א) " ערכים פשוטים ומצביעים לשני בנים " מצביע לאיבר הראשון בעץ 2. אם הינו מחליפים בפונקצית ההדפסה בין פקודת ההדפסה לקריאה השנייה של הפונקציה הינו מקבלים את האיברים לפי הסדר: (ג) " כן, מהקטן לגדול " כן, מהגדול לקטן " לא 3. מה סדר הכנסת האיברים בעץ? (ב) " בכל צומת הבן הימני גדול מהאב " בכל צומת הבן השמאלי קטן מהאב והבן הימני גדול " בכל צומת הבן השמאלי קטן מהאב 4. איזו מבין הפונקציות הבאות לא רקורסיבית? (א) " Set() " Del_t() " Display() " Add()

עצים בינאריים

עץ בינארי - מבנה נתונים מופשט הבנוי הצורה הפוכה לעץ טבעי כלומר שורשו למעלה והסתעפותו כלפי מטה, כך שבתחתית מצויים ענפיו ועליו.
הצומת העליון הוא שורש העץ. כל נקודת הסתעפות נקראת צומת ולכל צומת שני ענפים, שמאלי וימני (במקרה של עץ בינארי). שני צמתים של ענפים אלו נקראים בנים או ילדים (ימני ושמאלי) של צומת ההסתעפות וצומת ההסתעפות נקרא אב של שני בנים אלו.

הערה: העץ נקרא בינארי מכיוון שלכל צומת יש לכל היותר שני בנים. ניתן לייצר גם עצים תרינאריים ויותר על ידי הגדלת מספר הבנים לכל צומת.
כל צומת מהווה גם היא שורש של תת עץ (עץ נתונים הוא בעל דמיון עצמי מושלם).
לכל צומת יש הורה אחד בלבד.
עלה הוא צומת שאין לו ילדים, צומת פנימי הוא צומת שאינו עלה.
לצאצא (במסלול כל שני צמתים עוקבים מקיימים x אם קיים מסלול מצומת x צומת יקרא צאצא של יחס אב-בן).
שני צמתים יקראו אחים אם יש להם את אותו הורה.
גובה של צומת - אורך המסלול (מספר צמתים) מהצומת ועד לעלה.
גובה עץ - גובה של השורש.
גודל העץ - מספר צמתיו.
יתכן כי לצומת יהיה בן אחד בלבד.

צומת מכילה נתונים ושני מצביעים: אחד לבן ימני והשני לבן שמאלי.

מאחר וכל צומת הוא שורש של תת עץ - כל הפעולות על עצים הן רקורסיביות.
ערכו של בן ימני יהיה גדול מערכו של צומת האב וערכו של צומת שמאלי קטן מערכו של צומת האב.

לכן בהכנסת הנתונים נבדוק את הערך של הנתון שאנו מעוניינים להכניס לעץ - אם ערכו קטן מהשורש נפנה לבן השמאלי של השורש ואם לא נפנה לבן הימני של השורש.
כשנפעיל פעולה כזו ברקורסיה נתייחס כעת לצומת החדש כשורש של תת העץ שהתקבל. נשווה את הערך שאנו מבקשים להכניס לעץ עם השורש - אם קטן נפנה לבנו השמאלי ואם לא נפנה לבנו הימיני וכך הלאה עד אשר נמצא את המיקם המתאים לערך שלנו.

פעולות בסיסיות

init - אתחול העץ.
add, set - הכנסה לעץ.
del - מחיקת העץ.

תבנית של עץ:

```
struct tree
{
    type1 data1;
    type2 data2;
    ....
    struct tree* left;
    struct tree* right;
};
```

כמו ברשימה יש מצביע לעץ שמכיל את הכתובת של שורש העץ, כל צומת בעץ מכיל את השדות של המבנה tree ושני מצביעים לבנים.

add() הכנסת איבר לעץ

```
void add (struct tree** t, int x)
{
    if (*t == NULL)
        set (t, x);
    else if (x < (*t)->num)
        add (&(*t)->left, x);
    else
        add (&(*t)->right, x);
}
```

הכנסת האיבר מתבצעת על ידי פונקציה רקורסיבית add(). הפרמטרים שלה הם הערך שרוצים אם יש כמה כאלו יש לשלוח את) שהצומת (שמתפקדת בעץ כמו איברים שראינו ברשימות) תקבל הנתונים כפרמטרים לפונקציה add()) והפרמטר הנוסף - מצביע למצביע שמכיל את הכתובת של שורש העץ.
בהכנסת איבר חדש יש להתאים את מקומו על פי הערך שרוצים להכניס לעץ (על פי התבנית שכל

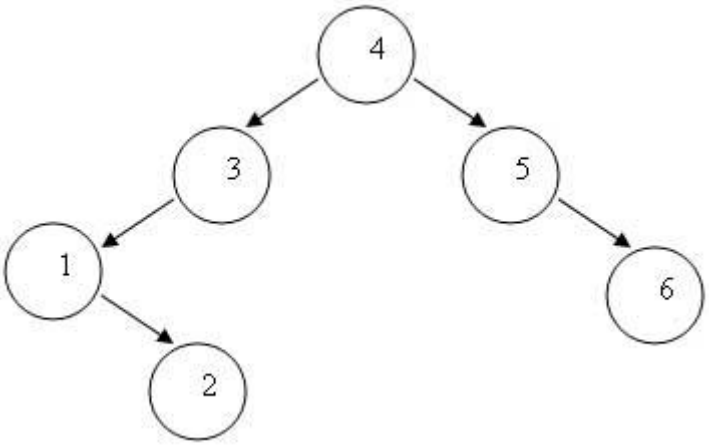
תת שורש מכיל בן שמאלי וימני כאשר הבן השמאלי מכיל ערך קטן מהשורש והבן הימני מכיל ערך גדול או שווה לשורש). כשמגיעים למקום המתאים, בונים צומת לפי הפונקציה set: הפונקציה set() מקבלת כתובת וערך, היא מקצה זיכרון לצומת החדש, מציבה את הערך המבוקש ומאתחלת את שני המצביעים של הצומת החדש ל- null - בעצם הפונקציה set כלומר הצומת החדש הוא עלה, בונה עלים לעץ.

הצגת אברי העץ display()

```
void display_t (struct tree* t)
{
    if (t != NULL)
    {
        display_t (t->left);
        printf ("%d\t", t->num);
        display_t (t->right);
    }
    else
        return;
}
```

גם הצגת איברים מתבצעת על ידי פונקציה רקורסיבית הפועלת כך:
לפונקציה נשלח האיבר הראשון של העץ. אם העץ לא ריק אז מציגים את תת עץ שמאלי - ניתן לראות זאת מכיוון שיש קריאה לפונקציה display() לאחר עם השורש של התת עץ השמאלי. עולים לאב של העלה ואז, (num) שהגענו לעלה השמאלי ביותר מדפיסים את השדה הפשוט שלו הפעם עם התת עץ הימני של הצומת שעליה דיברנו וכך הלאה יש שוב קריאה לפונקציה display() עם תת העץ הימני display() עד שמגיעים לשורש העץ, מדפיסים את ערכו ושוב קוראים לפונקציה של השורש.

עוברת על צמתי העץ display() להלן תרשים של עץ, נראה איך הפונקציה:
המספרים בצמתים הם הערכים שהצמתים מכילים - ערך השדה הפשוט (במקרה זה - שדה מספרי) במבנה האיבר מסוג צומת של עץ (הם לא מציינים את סדר הטיפול בצמתים או סדר ההכנסה או כל משמעות אחרת).



הפונקציה מתחילה בשורש - 4.
הצומת לא ריק לכן קוראים לפונקציה עם הצומת 3.
צומת 3 לא ריק ולכן קוראים לפונקציה עם הצומת 1.
הבן השמאלי של צומת 1 הוא null) הצומת 1 לא ריק ולכן קוראים לפונקציה עם null).
הצומת ריק וחוזרים לקריאה הקודמת עם צומת 1, מדפיסים את ערכו וקוראים לפונקציה עם 2.
הבן השמאלי של צומת 1 הוא null) הצומת לא ריק ולכן קוראים לפונקציה עם null).
הצומת ריק וחוזרים לקריאה הקודמת עם צומת 2, מדפיסים את ערכו וקוראים לפונקציה עם null.
הצומת ריק וחוזרים לקריאה הקודמת עם צומת 2, חוזרים לקריאה הקודמת עם צומת 1, חוזרים בנו הימיני של צומת) null לקריאה הקודמת עם צומת 3, מדפיסים את ערכו וקוראים לפונקציה עם 3).
הצומת ריק וחוזרים לקריאה הקודמת עם צומת 3, חוזרים לקריאה הקודמת עם צומת 4, מדפיסים את ערכו, וקוראים לפונקציה עם צומת 5.
בנו השמאלי של 5) הצומת לא ריק ולכן קוראים לפונקציה עם null).
הצומת ריק ולכן חוזרים לקריאה הקודמת עם צומת 5, מדפיסים את ערכו וקוראים לפונקציה עם הצומת 6.
הבן השמאלי של צומת 6) הצומת לא ריק ולכן קוראים לפונקציה עם).
הצומת ריק וחוזרים לקריאה הקודמת עם צומת 6, מדפיסים את ערכו וקוראים לפונקציה עם הצומת הבן הימיני של צומת 6) null).
הצומת ריק וחוזרים לקריאה הקודמת עם צומת 6, חוזרים לקריאה הקודמת עם צומת 5, חוזרים לקריאה הקודמת עם צומת 4, לעותק הראשון והפונקציה מסיימת.
נרשום רק את אשר הודפס למסך:
צומת 1, צומת 2, צומת 3, צומת 4, צומת 5, צומת 6 - יש לשים לב כי פונקצית הדפסה זו מדפיסה את האיברים לפי סדר מהקטן לגדול.

מחיקת העץ ()del_t

```
void del_t (struct tree** t)
{
    if (*t != NULL)
    {
        del_t (&(*t)->left);
        del_t (&(*t)->right);
        printf ("%p\t",*t);
        free (*t);
```

```
    }
    else
        return;
}
```

הפונקציה מקבלת את הכתובת של המצביע לשורש העץ. אם שורש העץ לא ריק קוראים לפונקציה עם הכתובת של השורש של תת העץ השמאלי, אם השורש של תת העץ השמאלי לא ריק שוב קוראים לפונקציה עם הכתובת של השורש של תת העץ של תת העץ של השורש של העץ וכך הלאה ולכן משחררים אותו על ידי פקודת delete. עד שמגיעים לעלה השמאלי ביותר. שני בניו הם null עולים בקריאות הרקורסיביות ושוב - אם תת העץ הימני לא ריק נכנסים לתת העץ הימני וכו' (התהליך של הקריאות הרקורסיביות דומה להצגת העץ בהבדל המזערי - רק לאחר שתי הקריאות הרקורסיביות מדפיסים את הכתובת ומשחררים את הזיכרון של הצומת). בסיום משחררים את שורש העץ.
נרשום בקצרה על פי העץ שלנו מהתרשים באיזה סדר משתחררים הצמתים:
צומת 2, צומת 1, צומת 3, צומת 6, צומת 5, צומת 4.
נאחד את כל הפונקציות לתוכנית אחת:

דוגמא 1:

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

const MAX = 7;

struct tree
{
    int num;
    struct tree* left;
    struct tree* right;
};

void init (struct tree** t)
{
    *t = NULL;
}

void set (struct tree** t, int x)
{
    *t = (struct tree*)malloc(sizeof(struct tree));
    (*t)->num = x;
    (*t)->right = NULL;
    (*t)->left = NULL;
}
```

```
void add (struct tree** t, int x)
{
    if (*t == NULL)
        set (t, x);
    else if (x < (*t)->num)
        add (&(*t)->left, x);
    else
        add (&(*t)->right, x);
}

void bulid_t (struct tree** t)
{
    int i, x;

    for (i = 0; i < MAX; i++)
    {
        x = random(30);
        add (t,x);
    }
}


void display_t (struct tree* t)
{
    if (t != NULL)
    {
        display_t (t->left);
        printf ("%d\t", t->num);
        display_t (t->right);
    }
    else
        return;
}


void del_t (struct tree** t)
{
    if (*t != NULL)
    {
        del_t (&(*t)->left);
        del_t (&(*t)->right);
        printf ("%p\t",*t);
        free (*t);
    }
    else
```

```
        return;
}

void main()
{
    struct tree* t;

    randomize();

    init (&t);
    bulid_t (&t);
    display_t (t);
    printf ("\nDeleting the tree in address %p :\n",&t);
    del_t (&t);
}
```

העבר את העכבר מעל הדוגמא כדי לראות הסבר מפורט

עץ מלא- עץ שלכל צומת בו יש שני בנים.
עץ שלם- עץ שבו לכל רמה i יש 2 i צמתים.

דוגמא 2:

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

struct tree
{
    int num;
    struct tree* left;
    struct tree* right;
};

void init (struct tree** t)
{
    *t = NULL;
}

void set (struct tree** t, int x)
{
    *t = (struct tree*)malloc(sizeof(struct tree));
    (*t)->num = x;
    (*t)->right = NULL;
```

```
    (*t)->left = NULL;
}

void add (struct tree** t, int x)
{
    if (*t == NULL)
        set (t, x);
    else if (x < (*t)->num)
        add (&(*t)->left, x);
    else
        add (&(*t)->right, x);
}
void bulid_t (struct tree** t)
{
    int i, x, max;
    max = 1 + random(5);
    for (i = 0; i < max; i++)
    {
        x = random(30);
        add (t,x);
    }
}

void display_t (struct tree* t)
{
    if (t != NULL)
    {
        printf ("%d\t", t->num);
        display_t (t->left);
        display_t (t->right);
    }
    else
        return;
}

int full_bin_t (struct tree** t)
{
    if (((*t)->left == NULL)&& ((*t)->right == NULL))
        return 1;
    else if (((*t)->left == NULL)|| ((*t)->right == NULL))
        return 0;
    else if (full_bin_t (&(*t)->left) && full_bin_t (&(*t)->right))
        return 1;
    else
        return 0;
}

void del_t (struct tree** t)
{
```

```
    if (*t != NULL)
    {
        del_t (&(*t)->left);
        del_t (&(*t)->right);
        printf ("%p\t", *t);
        free (*t);
    }
    else
        return;
}

void main()
{
    struct tree* t;
    randomize();

    init (&t);
    bulid_t (&t);
    display_t (t);
    if (full_bin_t (&t))
        printf ("\nThe tree is a full binary tree\n");
    else
        printf ("\nThe tree is not a full binary tree\n");
    del_t (&t);
}
```

הערה:
כאשר אנו שולחים כארגומנט את העץ יש לשלוח עם אופרטור הכתובת על מנת שהשינויים של הוספה לעץ ומחיקתו ישמרו אך כפרמטר הפונקציה יש להשתמש במצביע למצביע כך שהמצביע a88מכיל משתנה מסוג מצביע (המצביע לעץ). כך לדוגמא אם המצביע לעץ מכיל את הכתובת 0 אז המשתנה מצביע למצביע יהיה בכתובת מסוימת המכילה את fff4 וכתובתו של מצביע זה היא שתכיל את הכתובת של האיבר הראשון של העץ fff4 הכתובת.
כלומר מחוץ לפונקציות הנקראות:

```
&t = fff4
t = 0a88
*t האיבר הראשון של העץ.
בקריאה לפונקציות:
t = fff4
*t = 0a88
**t האיבר הראשון של העץ.
```

מבחן 1. שורש העץ מכיל: (א) " ערכים פשוטים ומצביעים לשני בנים " מצביע לאיבר הראשון בעץ 2.

בס"ד

אם הינו מחליפים בפונקצית ההדפסה בין פקודת ההדפסה לקריאה השנייה של הפונקציה הינו מקבלים את האיברים לפי הסדר: (ג) " כן, מהקטן לגדול " כן, מהגדול לקטן " לא 3. מה סדר הכנסת האיברים בעץ? (ב) " בכל צומת הבן הימני גדול מהאב " בכל צומת הבן השמאלי קטן מהאב והבן הימני גדול " בכל צומת הבן השמאלי קטן מהאב 4. איזו מבין הפונקציות הבאות לא רקורסיבית? (א)Add() " Display() " Del_t() " Set() "