פרק 15
מבנים ב-C

טיפוס חדש: מבנים

- **מבנה** (structure) הוא טיפוס מורכב בשפת C (בניגוד לטיפוס בסיסי).
- זהו טיפוס שמיועד לאיחוד קבוצת משתנים תחת שם אחד. משתנים אלו נקראים **השדות** של המבנה, והם יכולים להיות מטיפוסים שונים.
- מגדירים את קבוצת המשתנים בעזרת הפקודה **`struct`**.
- הפקודה **`struct`** משמשת להגדרת הטיפוס החדש בטרם משתמשים בו. היא יכולה להופיע בכל מקום בו יכולה להופיע פקודת **`typedef`**, כלומר בד"כ בתחילת הקובץ, או באזור ההצהרה על משתנים של פונקציה כלשהי.

טיפוס חדש: מבנים

- לדוגמה, נתבונן בהגדרה הבאה:

```
struct date {
  int day;
  int month;
  int year;
};
```

- הגדרה זו יוצרת טיפוס חדש בשם **`struct date`** המאגד בתוכו שלושה שדות מטיפוס **`int`**: **`day`**, **`month`** ו-**`year`**.
- שימו לב: פקודה זו אינה מקצה זיכרון כלשהו, אלא רק מגדירה טיפוס חדש. טיפוס זה יוכל לשמש אותנו בהמשך להצהרה על משתנים.

שימוש במבנים

- נגדיר כעת מספר משתנים מטיפוס **`struct date`**. כל משתנה כזה יכיל בתוכו שלשה של תתי-משתנים, או שדות, מטיפוס **`int`**:

```
struct date ilan_birthday, neta_birthday;
```

- על מנת לגשת ל-**`int`** כלשהו בתוך **`neta_birthday`**, עלינו להשתמש באופרטור נקודה (**`'.'`**):

```
neta_birthday.day = 5;
neta_birthday.month = 2;
neta_birthday.year = 1982;
```

שימוש במבנים

- שימו לב שאין בלבול בין השדה **day** של מבנים שונים, שכן יש צורך להשתמש באופרטור נקודה **'.'** על מנת לגשת למשתנה הפנימי של מבנה כלשהו. כלומר, אנו תמיד מציינים במפורש את שם המבנה אליו שייך השדה **day** הרצוי.
- למשל, נוכל לכתוב:

```
printf("%d\n", ilan_birthday.day);
```

- אבל השורה הבאה לא תעבור קומפילציה, שכן לא מוגדר משתנה עצמאי כלשהו ששמו '**day**' :

```
printf("%d\n", day);
```


שימוש במבנים

- ניתן לבצע השמה של מבנה כלשהו למבנה אחר מאותו הטיפוס. פעולת ההשמה **מעתיקה** את התוכן של כל אברי המבנה לתוך אלו של המבנה השני. לדוגמה:

```
struct date twin1_birthday;
struct date twin2_birthday;

twin1_birthday.day = 30;
twin1_birthday.month = 12;
twin1_birthday.year = 1999;

twin2_birthday = twin1_birthday;
```

- פעולת ההשמה בשורה האחרונה מבצעת העתקה של שלושה **int**-ים, מהמבנה **twin1_birthday** למבנה **twin2_birthday**.

תוכנית לניהול משתלה

- מבנה יכול להכיל משתנים מטיפוסים שונים. לדוגמה, נתבונן במבנה שמייצג צמח במשתלה:

```
enum season {summer, fall, winter, spring};

struct plant
{
  char name[100];
  double water_per_day;
  double fertilizer_per_day;
  enum season flower_season;
  struct date plant_date;
};
```

- **struct date** שהגדרנו קודם הוא עצמו טיפוס לכל דבר, ולכן ניתן לכלול גם משתנה מטיפוס זה בטיפוס החדש **struct plant**.

קריאת הפרמטרים של צמח

- נכתוב תוכנית שקוראת מהמשתמש את הפרמטרים של צמח כלשהו:

```
int main()
{
  struct plant p;
  printf("Welcome to PlantRage Xtreme!\n");

  printf("Please enter your plant's name: ");
  scanf("%s", p.name);

  printf("Enter liters of water per day: ");
  scanf("%lf", &p.water_per_day);

  …
}
```

הדפסת הפרמטרים של צמח

- השורה הארוכה הבאה מדפיסה את הפרמטרים של הצמח:

```
printf("Plant specs for plant %s:\n"
       "it needs %lf liters of water per day\n"
       "and %lf kgs of fertilizer per day.\n"
       "It was planted on %d.%d.%d\n"
       "and flowers in the %s!\n",

       p.name, p.water_per_day,
       p.fertilizer_per_day,
       p.plant_date.day, p.plant_date.month,
       p.plant_date.year,
       (p.flower_season==0) ? "summer" :
       (p.flower_season==1) ? "fall" :
       (p.flower_season==2) ? "winter" : "spring"
);
```


צמח לדוגמה

- ריצה לדוגמה של התוכנית:

```
Welcome to PlantRage Xtreme!
Please enter your plant's name: GreatPlant
Enter liters of water per day: 5.3
Enter kgs of fertilizer per day: 1.8
Enter flowering season
(0-summer, 1-fall, 2-winter, 3-spring): 3
Enter plant date (d.m.y): 5.7.2001

Plant specs for plant GreatPlant:
it needs 5.300000 liters of water per day
and 1.800000 kgs of fertilizer per day.
it was planted on 5.7.2001
and flowers in the spring!
```

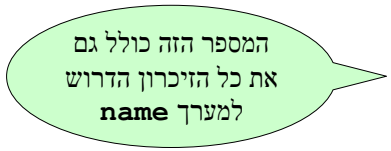

אופרטור sizeof למבנים

- ניתן להפעיל את האופרטור **sizeof** על מנת לקבל את כמות הזיכרון שצורך מבנה כלשהו. כמות הזיכרון שהוא צורך שווה לסך הזיכרון שדורשים כל השדות שהוא כולל.
- למשל, השורות הבאות:

```
printf("%d\n", sizeof(struct date));
printf("%d\n", sizeof(ilan_birthday));
printf("%d\n", sizeof(struct plant));
```

- יגרמו לפלט הבא:

```
6
6
124
```



מערך של מבנים

- כזכור ניתן ליצור ב-C מערך מכל טיפוס. כיוון שמבנים הם בוודאי טיפוסים, נוכל ליצור מהם מערכים כרגיל:

```
struct date birthday_list[10];
```

- נוכל להשתמש במערך כרגיל:

```
birthday_list[3].day = 5;
scanf("%d", &birthday_list[3].month);
birthday_list[3].year ++;
```

- גודלו של מערך זה הוא **10*sizeof(struct date)**.

מבנים ומצביעים

- כיוון שכבר הגענו עד מערכים, דרכנו קצרה להגדרת מצביעים:

```
struct plant *plant_ptr = 0;
```

- המשתנה **plant_ptr** הוא מצביע שמכיל כתובות של משתנים מטיפוס **struct plant**.
- אתחלנו את **plant_ptr** ל-0 (null) על מנת לציין שהוא איננו מצביע כרגע לשום כתובת חוקית בזיכרון.
- נוכל לכוון את **plant_ptr** להצביע על צמח מסוים. הגישה לצמח המוצבע תעשה כרגיל על ידי האופרטור * :

```
struct plant p;
plant_ptr = &p;
(*plant_ptr).water_per_day = 30.0;
```


struct כפרמטר לפונקציה

- ניתן להעביר כל מבנה כפרמטר לפונקציה. ההעברה מתבצעת by-value, כלומר כל השדות הפנימיים של המבנה מועתקים בשלמותם לפרמטר של הפונקציה. ההעתקה כוללת מערכים (!), שתוכנם מועתק בשלמותו.
- לכן, כל שינוי שנבצע בתוך הפונקציה לא ישפיע על המבנה המקורי ששלחנו אליה.
- לדוגמה, הפונקציה הבאה מדפיסה תאריכים מהטיפוס **struct date**:

```
print_date(struct date d) {
  printf("%d.%d.%d", d.day, d.month, d.year);
}
```


struct כפרמטר לפונקציה

- נוכל גם לכתוב פונקציה שמדפיסה את הפרמטרים של צמח, כך:

```
void disp_plant(struct plant p)
{
  printf("Plant specs for plant %s:\n"
         "it needs %lf liters of water per day\n"
         "and %lf kgs of fertilizer per day.\n"
         ...
  );
}
```

- אבל, שימו לב! במקרה של צמח, הפונקציה בעייתית מאוד, שכן כל קריאה אליה דורשת העתקה של כל תוכן המבנה לפרמטר הפנימי **p** של הפונקציה, ואלו 124 בתים שאנו מעתיקים רק לצורך הדפסה.

struct כפרמטר לפונקציה

- בגלל תכונת ההעתקה by-value, נהוג בדרך כלל לכתוב פונקציות שמקבלות מצביעים למבנים ולא את המבנים עצמם. בצורה זו חוסכים זמן העתקה מיותר, וגם חוסכים זיכרון.
- נשכתב לכן את הפונקציה להדפסת הצמח כך:

```
void disp_plant(struct plant *p)
{
  printf("Plant specs for plant %s:\n"
         "it needs %lf liters of water a day\n"
         "and %lf kgs of fertilizer a day.\n"
         …
         (*p).name, …
  );
}
```

struct כפרמטר לפונקציה

- כעת נוכל גם לכתוב פונקציה שקוראת צמח חדש מהמשתמש, שכן העברת מצביע למבנה מאפשרת לה לבצע שינויים במבנה המקורי שנשלח אליה.

```
void read_plant(struct plant *p)
{
  printf("Please enter your plant's name: ");
  scanf("%s", (*p).name);

  printf("Enter liters of water per day: ");
  scanf("%lf", &(*p).water_per_day);
  …
}
```


הכל ביחד: מבנים, מצביעים ופונקציות

- התוכנית הבאה קוראת מהמשתמש את מס' הצמחים שברשותו, וקוראת את הפרמטרים שלהם:

```
struct plant *plants;
int i, n;

printf("Welcome to PlantRage Xtreme!\n");
printf("Enter number of plants: ");
scanf("%d", &n);

plants = (struct plant*)
  malloc(n*sizeof(struct plant));
for (i=0; i<n; ++i)
  read_plant(plants+i);
```


נקודות במישור

- מטרתנו כעת תהיה לכתוב אוסף פונקציות שמבצעות חישובים גיאומטריים במישור.
- לשם כך נתחיל בהגדרת מבנה שייצג לנו נקודה במישור הדו-ממדי:

```
struct point_2d {
  double x, y;
};
```

- פונקציה להדפסת הנקודה תראה כך:

```
void disp_point(struct point_2d *p) {
  printf("(%lf,%lf)", (*p).x, (*p).y);
}
```


typedef כמה קיצורי דרך (1):

- אנו יכולים להשתמש ב-**typedef** על מנת לתת שם קצר יותר לטיפוס **struct point_2d**. זה נעשה באופן הבא:

```
struct point_2d {
  double x, y;
};
typedef struct point_2d point2d;
```

- יש גם הנוהגים לקצר ולאחד את שתי השורות לשורה יחידה כך:

```
typedef struct {
  double x, y;
} point2d;
```

- בשתי השיטות התוצאה היא אחת: אנו יכולים מעתה להתייחס לטיפוס **point2d** (ללא המילה **struct**) כשם מקוצר לטיפוס המקורי.

כמה קיצורי דרך (2): האופרטור ->

- כיוון שעבודה עם מצביעים למבנים כל כך נפוצה בשפת C, השפה מגדירה אופרטור מיוחד שמשמש עבור מצביעים למבנים, והופך את העבודה עימם לנוחה יותר.
- האופרטור **->** (חץ) שמופעל על מצביע למבנה, מאפשר גישה ישירה לתוך השדות של המבנה המוצבע. למשל, נתבונן בהגדרה:

```
point2d point;
point2d *point_ptr = &point;
```

- בעזרת המצביע **point_ptr**, נוכל כעת לגשת לשדות הפנימיים של המבנה **point** בשתי דרכים (שקולות לחלוטין):

```
(*point_ptr).x
(*point_ptr).y
```

⟺

```
point_ptr -> x
point_ptr -> y
```


בחזרה למישור

- בעזרת שני קיצורי הדרך, הפונקציה להדפסת נקודה נרשמת כעת כך:

```
void disp_point(point2d *p) {
  printf("(%lf,%lf)", p->x, p->y);
}
```

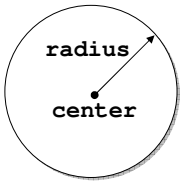
- נוכל להוסיף גם פונקציה הקוראת נקודה מהמשתמש:

```
void read_point(point2d *p) {
  scanf("( %lf , %lf )", &p->x, &p->y);
}
```


אובייקטים נוספים במישור

- נוסיף כעת כמה אובייקטים למישור הדו-ממדי שלנו.
- נייצג מעגל ע"י נקודת המרכז שלו, והרדיוס שלו:

```
struct circle {
  point2d center;
  double radius;
};
typedef struct circle circle;
```

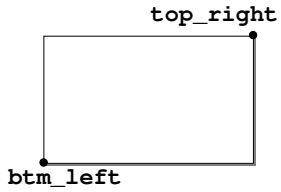


- הערה: השורה האחרונה איננה טעות, שפת C מאפשרת לתת לטיפוס "**struct circle**" את השם הנרדף "**circle**".

אובייקטים נוספים במישור

- נייצג מלבן ע"י הנקודה השמאלית-תחתונה שלו והנקודה הימנית-עליונה שלו:

```
struct rect {
  point2d btm_left;
  point2d top_right;
};
typedef struct rect rect;
```



- ניתן להגדיר באותו אופן צורות נוספות, כמו קו, משולש, אליפסה ועוד. כרגע נסתפק בדוגמאות האלה.

פונקציות לטיפול בגופים הגיאומטריים

- על מנת להפוך את ההגדרות הללו לבעלות משמעות, נרצה לכתוב מספר פונקציות שידעו להשתמש בטיפוסים החדשים.
- נתחיל מפונקציה שמחשבת מרחק בין שתי נקודות. כזכור, המרחק בין שתי נקודות (x_1,y_1), (x_2,y_2) נתון על ידי:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- פונקציה המממשת נוסחה זו היא:

```
double dist(point2d *p1, point2d *p2) {
  return sqrt( pow(p2->x - p1->x, 2) +
               pow(p2->y - p1->y, 2) );
}
```


פונקציות לטיפול בגופים הגיאומטריים

- פונקציות שמחשבות שטח:

```
double area(circle *c) {
  return PI * c->radius * c->radius;
}
```

```
double area(rect *r) {
  return (r->top_right.x - r->btm_left.x) *
         (r->top_right.y - r->btm_left.y) ;
}
```


פונקציות לטיפול בגופים הגאומטריים

- הפונקציה **`point_in_rect()`** בודקת אם נקודה נמצאת בתוך מלבן כלשהו.
- לשם כך יש לוודא שה-x של הנקודה נמצא בין ערכי ה-x של המלבן, ושה-y של הנקודה נמצא בין ערכי ה-y של המלבן:

```
int point_in_rect(rect *r, point2d *p)
{
  return ( (p->x >= r->btm_left.x) &&
           (p->x <= r->top_right.x) &&
           (p->y >= r->btm_left.y) &&
           (p->y <= r->top_right.y) );
}
```


פונקציות לטיפול בגופים הגאומטריים

- הפונקציה **`point_in_circ()`** בודקת האם נקודה נמצאת בתוך עיגול כלשהו.
- לשם כך יש לוודא שהמרחק בין הנקודה לבין מרכז המעגל קטן מרדיוס המעגל:

```
int point_in_circ(circ *c, point2d *p)
{
  return (dist(&c->center, p) <= c->radius);
}
```

פונקציות לטיפול בגופים הגאומטריים

- הפונקציה **`rect_in_rect()`** בודקת האם המלבן **`r2`** נמצא במלואו בתוך המלבן **`r1`**.
- לשם כך יש לוודא ששתי הפינות של המלבן האחד נמצאות שתיהן בתוך גבולות המלבן השני:

```
int rect_in_rect(rect *r1, rect *r2)
{
  return (point_in_rect(r1, r2->btm_left) &&
          point_in_rect(r1, r2->top_right));
}
```


פונקציות לטיפול בגופים הגאומטריים

- הפונקציה הבאה בודקת אם מעגל כלשהו נמצא במלואו בתוך מלבן מסוים. לשם כך היא מוודאת שהמלבן החוסם את המעגל נמצא במלואו בתוך המלבן הנתון.

```
int circ_in_rect(rect *r, circ *c)
{
  rect r2; // rectangle enclosing the circle
  r2.btm_left = r2.top_right = c->center;
  r2.btm_left.x -= c->radius;
  r2.btm_left.y -= c->radius;
  r2.top_right.x += c->radius;
  r2.top_right.y += c->radius;

  return rect_in_rect(r, &r2);
}
```


פונקציות לטיפול בגופים הגאומטריים

- אתם מוזמנים להמשיך ולממש בעצמכם מבנים ופונקציות נוספות בספרייה !