פרק 3
מזהים, טיפוסים ואופרטורים

מזהים (identifiers)

- המתכנת נותן שמות (=מזהים) לאלמנטים שונים בתוכנית: משתנים, פונקציות, ועוד.
- ישנם כללים לגבי אילו שמות ניתן לתת.
- הכללים הללו אינם תלויים בסוג האלמנט.
- כדאי, מומלץ וחובה לתת שמות בעלי משמעות!

כללים לגבי מזהים

- ניתן להשתמש אך ורק באותיות אנגליות (קטנות וגדולות), ספרות, והתו '_' (underscore).
- התו הראשון בשם אינו יכול להיות ספרה.
- אורך המזהה אינו מוגבל. עם זאת, ישנם קומפיילרים המבחינים בין מזהים רק על פי 31 התווים הראשונים שלהם. כלומר, אם שני מזהים שונים מתלכדים ב-31 התווים הראשונים שלהם, הקומפיילר יתייחס אליהם כאל אותו מזהה.
- ישנה רשימה של שמות שבהם אסור להשתמש, כיוון שאלו מילים המשמשות את השפה עצמה. אלו מכונות מילים שמורות.

מילים שמורות

- כמה מאלה אתם כבר מכירים?

`auto`	`break`	`case`	`char`
`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`

דוגמאות למזהים

- מזהים חוקיים:

```
temp          counter
the_one       intro2cs
```

- מזהים לא חוקיים:

`1st_street` : מתחיל בספרה

`do_it!` : תו לא חוקי '!'

`drink-me` : תו לא חוקי '-'

הנחיות וטיפים לגבי שימוש בשמות

- שפת C מבחינה בין אותיות קטנות וגדולות.

```
int the_num, the_Num;
double Double;
```

- לא מומלץ 'לדרוס' שמות שכבר יש להם שימוש בשפה. זה יוצר קוד מבלבל, וגם חוסם למתכנת את הגישה למה שהשמות הללו ייצגו קודם.

```
double main;
int printf;
```


הנחיות וטיפים לגבי שימוש בשמות

- שמות משתנים ופונקציות נהוג לכתוב באותיות קטנות.

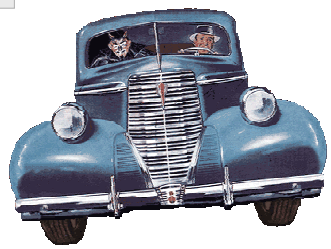
```
int index;
double num;
```

- שמות שמוגדרים ב- `#define` נהוג לכתוב באותיות גדולות.

```
#define PI 3.14159
#define INT_MAX 32767
```

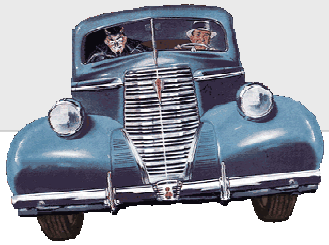

מה עושה התוכנית הבאה:

```
cst = wh * pphr + prt;

pt = cst * (1 + v);
```



ועכשיו?

```
total_cost =
  work_hours * price_per_hour + parts_cost;

payment =
  total_cost * (1 + vat);
```



משתנים

- אנו משתמשים בתאי זיכרון על מנת לאחסן נתונים, תוצאות, וערכי ביניים.
- השימוש בזיכרון נעשה באמצעות מנגנון של **משתנים**.

- **משתנה**: קבוצת תאי זיכרון שאנו נותנים לה שם, ומשמשת לאגירת נתון אחד.
- בתחילת התוכנית יש להצהיר על כל משתנה שבו נשתמש, על מנת שהתוכנית תקצה לו את הזיכרון הדרוש.

שימוש במשתנים

- יש להצהיר על כל המשתנים בתחילת התוכנית, מייד לאחר הסוגר הפותח של **main()**.
- ניתן להגדיר כמה משתנים מאותו הטיפוס יחד, על ידי הפרדתם בפסיקים.
- ניתן לתת לכל משתנה ערך התחלתי (= אתחול המשתנה). ערך זה יוכנס למשתנה מייד כאשר הוא מוקצה.
- משתנה שאיננו מאותחל עשוי להכיל כל ערך שהוא ("זבל").

שימוש במשתנים

```
int main()
{
  int idan;
  double dudu;
  char coral;

  double average_grade = 0;
  int sum = 0;
  …
}
```

משתנים אלו אינם מאותחלים, ולכן מכילים ערכים אקראיים בתחילת התוכנית.

משתנים אלו מאותחלים, ולכן אנו יודעים עם תחילת התוכנית מה יהיה ערכם.

טיפוסים בסיסיים ב-C

- **`int`**
  קיצור של: integer
  מייצג: מספר שלם
- **`double`**
  קיצור של: double precision
  מייצג: מספר עשרוני "ממשי"
- **`char`**
  קיצור של: character
  מייצג: מספר שלם קטן, או תו בודד

מספרים שלמים

מספרים שלמים ניתן לייצג באמצעות שלושה טיפוסים:

`short`	`int`	`long`

ההבדל ביניהם הוא במספר הבתים המוקצים לייצוג המספר.

- מספר הבתים המוקצה לכל טיפוס אינו מוגדר בשפה, ועשוי להשתנות מקומפיילר אחד לשני.
- מובטח שמספר הבתים המוקצה לטיפוסים אלה מקיים:

```
short  ≤  int  ≤  long
```

- הרעיון הוא ש- **`int`** יהיה בגודל המילה הטבעית של המעבד (... אבל זה לא תמיד קורה). מס' הבתים במקרה הטיפוסי –

```
short : 2      int : 2/4      long: 4
```


טווח הייצוג של מספרים שלמים

- 2 בתים (= 16 ביטים):

$$-2^{15} \le X \le 2^{15} - 1 \qquad 2^{15} \approx 32{,}000$$

- 4 בתים (= 32 ביטים):

$$-2^{31} \le X \le 2^{31} - 1 \qquad 2^{31} \approx 2 \text{ billion}$$

מספרים שלמים ללא סימן

- בייצוג הרגיל של המספרים השלמים, משתמשים באחד הביטים על מנת לייצג את הסימן של המספר.
- על ידי הוספת המילה **`unsigned`** לפני שם הטיפוס, אנו מוותרים על הייצוג של המספרים השליליים.
- במקום זאת אנו מגדילים את טווח הייצוג של המספרים החיוביים פי 2.
- אנו מקבלים את הטיפוסים החדשים הבאים:

```
unsigned short
unsigned int
unsigned long
```

מספרים שלמים ללא סימן

- 2 בתים (= 16 ביטים):

$$0 \le X \le 2^{16} - 1 \qquad 2^{16} \approx 65{,}000$$

- 4 בתים (= 32 ביטים):

$$0 \le X \le 2^{32} - 1 \qquad 2^{32} \approx 4 \text{ billion}$$

קבועים שלמים

- כל קבוע שלם (=מספר מפורש) בתוכנית נחשב מטיפוס **int**.
- ניתן להוסיף סיומת לקבוע בתוכנית על מנת לגרום לקומפיילר להתייחס אל הקבוע כבעל טיפוס אחר מאשר **int**:

דוגמאות	הטיפוס של הקבוע	הסיומת
`5000 , -30`	`int`	
`256U , 30u`	`unsigned int`	`u` או `U`
`256L , 30l`	`long`	`l` או `L`
`1234567UL`	`unsigned long`	`ul` או `UL`

מספרים ממשיים

מספרים ממשיים ניתן גם כן לייצג בעזרת שלושה טיפוסים:

```
float      double      long double
```

ההבדל ביניהם הוא כרגיל במספר הבתים המוקצים לייצוג המספר.

- מספר הבתים המוקצה לכל טיפוס איננו מוגדר בשפה, ועשוי להשתנות מקומפיילר אחד לשני.
- מובטח שמספר הבתים המוקצים לטיפוסים אלה מקיים:

```
float  ≤  double  ≤  long double
```

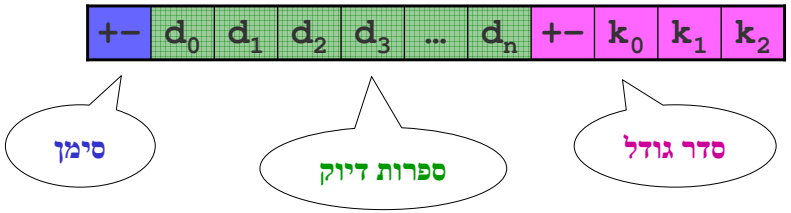
- מספר הבתים המוקצים במקרה הטיפוסי הוא:

```
float:4, double:8, long double: 8/12/16
```


מספרים ממשיים

- מספר ממשי מיוצג במחשב באופן הבא:

$$\pm d_0 . d_1 d_2 d_3 \ldots d_n \cdot 10^{\pm k}$$



מספרים ממשיים - דוגמאות

- כל מספר ממשי ניתן להביא לצורה זו:

1500	→	$+1.5 \cdot 10^{3}$
20.5	→	$+2.05 \cdot 10^{1}$
0.005	→	$+5.0 \cdot 10^{-3}$
-3.141	→	$-3.141 \cdot 10^{0}$

יכולת הייצוג של מספרים ממשיים

מספר הבתים בייצוג משפיע על:

- מספר ספרות הדיוק (=עד כמה המספר מדויק)
- טווח סדרי הגודל (=עד כמה המספר יכול להיות קטן או גדול)

4 בתים:

- כ-6 ספרות דיוק עשרוניות
- טווח סדרי הגודל הוא בקירוב: $10^{-38}...10^{+38}$

8 בתים:

- כ-15 ספרות דיוק עשרוניות
- טווח סדרי הגודל הוא בקירוב: $10^{-308}...10^{+308}$

קבועים ממשיים

- בשפת C ניתן לציין מספרים ממשיים בשתי צורות: בכתיבה רגילה ובכתיבה מדעית.
- דוגמאות לכתיבה רגילה: .99 , -24.000 , 1.56
- בכתיבה מדעית מצרפים למספר הממשי סיומת הכוללת את האות e ולאחריה מספר שלם. המספר השלם מציין הכפלה של המספר הממשי שציינו בחזקה זו של 10. דוגמאות:

0.005	→	5e-3
-3.14	→	-0.314e1

קבועים ממשיים

- כל קבוע ממשי בתוכנית נחשב מטיפוס **double**.
- ניתן להוסיף סיומות לקבוע על מנת לגרום לקומפיילר להתייחס אל הקבוע כבעל טיפוס אחר:

הסיומת	הטיפוס של הקבוע	דוגמאות
	`double`	`1.5 , -3.0` `4.2e5 , 10e-60`
`f` או `F`	`float`	`36.7F , 4.2e+5f`
`l` או `L`	`long double`	`36.7L , .51`

ייצוג תווים

- בשפת C מבחינים בין 256 תווים שונים. לכל תו מתאים ייצוג בינארי בעל 8 ביטים, כלומר מספר שלם בטווח 0…255.
- ישנה טבלה המתאימה לכל תו את המספר המייצג אותו, והיא נקראת **טבלת ASCII** -
American Standard Code for Information Interchange

תכונות שימושיות של הטבלה:

- הערכים המתאימים לתווים '0'…'9' הם רציפים, ולפי סדר הספרות.
- הערכים המתאימים לתווים 'a'..'z' רציפים, ולפי סדר הא"ב הרגיל.
- כך גם הערכים המתאימים לתווים 'A'…'Z'.

קטע מטבלת ASCII

תו	קוד ASCII	תו	קוד ASCII
'?'	63	'7'	55
'@'	64	'8'	56
'A'	65	'9'	57
'B'	66	':'	58
'C'	67	';'	59
'D'	68	'<'	60
'E'	69	'='	61

ייצוג תווים

- תווים מיוצגים בזיכרון כמספר המציין את ערך ה-ASCII שלהם.
- ניתן לייצג תו באמצעות כל טיפוס של מספר שלם.
- על מנת לראות את התו עצמו, יש להשתמש בתג `%c` בתוך `printf()`.

```
int ch = 65;
printf("%d\n", ch);
printf("%c\n", ch);
```

הפלט:

```
65
A
```


קבועי תווים

- על מנת לאחסן תו כלשהו במשתנה, יוצא שעלינו לברר בכל פעם את ערכו המספרי של התו בטבלת ASCII, ולהכניס ערך זה למשתנה. למשל, האות A מיוצגת על ידי המספר 65:

```
int ch = 65;
```

- בפועל, שיטה זו מאוד לא נוחה כיוון שהיא דורשת שינון של כל הטבלה בעל פה, או לחילופין בדיקה בטבלת ASCII בכל פעם.
- חסרון חמור מזה הוא ששפת C כלל אינה מחייבת שימוש בטבלת ASCII, וייתכן קומפיילר שבו טבלת התווים שונה לחלוטין! למשל, בקומפיילר המשתמש בטבלת EBCDIC, האות A מיוצגת על ידי המספר 193...

קבועי תווים

- שפת C פותרת בעיה זו באמצעות המנגנון של קבועי תווים. במקום לציין את מספרו של התו בקוד, נכתוב מפורשות את התו הרצוי, בתוך זוג גרשיים בודדים:

'a'	'b'	'A'	'5'	'+'	'?'	' '	'\n'

- הקומפיילר מתייחס לקבועי התווים הללו כמספרים לכל דבר. למעשה, מבחינת הקומפיילר סימונים אלה הינם פשוט שמות נרדפים למספרים שלמים מטיפוס **int**!

```
int x = 'A' + 46;   ⟷   int x = 65 + '.';
```

- שימוש בקבועי תווים מונע את התלות של הקוד שלנו בטבלת התווים בה הקומפיילר משתמש. יש להשתמש בקבועי תווים בלבד לציון תווים.

דוגמאות לעבודה עם תווים

```
printf("%c = %d", 97, 97);
```

```
a = 97
```

```
printf("%c = %d", 'b', 'b');
```

```
b = 98
```

```
printf("%c%c%c",
       'a', 'a'+1, 'a'+2);
```

```
abc
```


הטיפוס char

- **char** הינו טיפוס של מספר שלם, המותאם במיוחד לייצוג תווים.
- **char** מאוחסן בבית אחד, ולכן הוא מסוגל לייצג בדיוק את כל הערכים בטבלת ASCII (את כל התווים האפשריים).
- **char הוא ראשית לכל טיפוס של מספר שלם**, והיחס אליו הוא כמו אל כל טיפוס שלם אחר. ניתן לבצע עליו כל פעולה שניתן לבצע על טיפוסים שלמים אחרים.

- כמספר שלם, טווח הייצוג של **char**: $-128 \le \mathbf{X} \le 127$
- ויש גם **unsigned char**: $0 \le \mathbf{X} \le 255$

הטיפוס char

- כמשתנה לייצוג מספרים שלמים, הטווח של **char** קטן מאוד ולכן לרוב איננו שימושי במיוחד.
- השימוש הנפוץ ב-**char** הוא אכן לייצוג תווים מטבלת ASCII.
- כאשר מייצגים תווים, אין הבדל בין שימוש ב-**char** לשימוש ב-**unsigned char**. שתי האפשרויות ייתנו אותה התנהגות.

```
char ch = 'A';
printf("%d\n", ch);
printf("%c\n", ch);
```

הפלט:

```
65
A
```

דוגמאות לשימוש ב-char (1)

```
char c = 'a';
while (c <= 'z') {
  printf("%d: %c\n",c,c);
  c=c+1;
}
```

Run!

```
97: a
98: b
99: c
100: d
…
122: z
```


דוגמאות לשימוש ב-char (2)

- מה הפלט של התוכנית הבאה ?

```
char letter;

printf("Enter a lowercase letter: ");
scanf("%c", &letter);

printf("In uppercase: %c",
       (letter - 'a') + 'A');
```

Run!

המרת משתנים אוטומטית

- בשפת C ישנן פעולות רבות המערבות שני איברים:

  a>b a+=b a=b a*b a+b

- אם האיברים אינם מאותו הטיפוס, מתבצעת <u>המרה אוטומטית</u>.
- עבור פעולות השמה, הערך בצדו הימני של האופרטור מומר לטיפוסו של המשתנה שלתוכו כותבים.
- לכל יתר האופרטורים, הטיפוס עם תחום הייצוג הקטן יותר מומר לזה עם תחום הייצוג הגדול יותר, לפי הסדר הבא:

char → short → int → long →
float → double → long double

מסיבת מיץ!

```
int apples = 30, children = 12;
double juice_from_apple = 0.1;
double tot_orange_juice = 5.4;
double liters_per_cup = 0.3;
```

- במסיבה שלנו מחולק מיץ לכולם!
- יש קנקן מיץ תפוזים מוכן, וכן תפוחים שאותם נחלק לילדים על מנת שיסחטו את המיץ בעצמם.
- עלינו לחלק הכול שווה בשווה. אולם את התפוחים איננו רוצים לחתוך במסיבה, ולכן עשויים להישאר מהם עודפים.

מסיבת מיץ!

```
int apples = 30, children = 12;
double juice_from_apple = 0.1;
double tot_orange_juice = 5.4;
double liters_per_cup = 0.3;
```

```
double orange_juice_per_child =
   tot_orange_juice / children;
```

0.45

5.4 (double)

12 (int)

5.4/12 חילוק רגיל

מסיבת מיץ!

```
int apples = 30, children = 12;
double juice_from_apple = 0.1;
double tot_orange_juice = 5.4;
double liters_per_cup = 0.3;
```

```
double apple_juice_per_child =
  (apples / children) * juice_from_apple;
```

0.2

30/12 חילוק בשלמים

0.1 (double)

2*0.1 (double)

מסיבת מיץ!

```
int apples = 30, children = 12;
double juice_from_apple = 0.1;
double tot_orange_juice = 5.4;
double liters_per_cup = 0.3;
```

```
double total_juice_per_child =
   orange_juice_per_child +
   apple_juice_per_child;
```

0.65

0.2 (double)

0.45 (double)

0.45+0.2 (double)

מסיבת מיץ!

```
int apples = 30, children = 12;
double juice_from_apple = 0.1;
double tot_orange_juice = 5.4;
double liters_per_cup = 0.3;
```

```
int whole_cups_per_child =
  total_juice_per_child / liters_per_cup;
```

2

0.65/0.3 = 2.16667 (double)

2.1667 → 2 המרה ל-int

המרה אוטומטית – סוף הסיפור?

- המרה אוטומטית אינה מתבצעת כשמשתמשים בפונקציה **printf()** ודומותיה.
- לכן, אם נעביר לפונקציה משתנה מטיפוס שאיננו מתאים לתו הבקרה, אנו עשויים לקבל תוצאות בלתי צפויות.
- במקרה זה יש לבצע <u>המרה מכוונת</u>.
- ומה לגבי התוכנית הבאה?

```
short num1, num2;
long product;
product = num1 * num2;
```

מדוע זה עלול לא לעבוד?

המרה מכוונת (casting)

- המרת ערך כלשהו לטיפוס אחר נעשית על ידי כתיבת שם הטיפוס החדש בסוגריים, לפני הערך עצמו. למשל :

```
int x = 2;
double d;
printf("%lf", (double)x );
d = (double)3 / 2;
```

מחזיר את ערכו של **x** כטיפוס **double**

- פעולת ההמרה לוקחת את הערך הנתון, ומחזירה **עותק שלו** מהטיפוס החדש (היא אינה משנה את הערך המקורי!).
- בדוגמה למעלה, **x** עצמו <u>איננו משתנה</u> כתוצאה מהפעולה.

המרות בין טיפוסים

- עבור טיפוסים בסיסיים, שפת C יודעת לבצע <u>המרה חכמה</u> ביניהם. כלומר, תוצאת ההמרה הינה אותו הערך שהומר (עד כמה שאפשר) – רק בטיפוס החדש.
- פעולת ההמרה עשויה לאבד אינפורמציה כיוון שלטיפוסים שונים יכולת ייצוג שונה.
- בשפת C ניתן להמיר כמעט <u>כל טיפוס לכל טיפוס אחר</u>. לטיפוסים עבורם לא מוגדרת המרה חכמה, מתבצעת המרה "ברוטלית", כלומר פשוט מתייחסים לאותו אזור בזיכרון כאילו הוא מייצג משהו אחר.
- באופן כללי, עבור טיפוסים שלא מוגדרת בשבילם המרה חכמה, המרה מכוונת מטיפוס אחד לטיפוס שונה לחלוטין היא פעולה שיש להשתמש בה בזהירות רבה.

דוגמאות להמרות בין טיפוסים שונים

int → long	לא נאבד דבר, כי ממירים לטיפוס גדול יותר.
long → int	אם הערך בתוך הטווח של **int**, לא נאבד דבר, אולם אם הערך מחוץ לטווח של **int**, התוצאה לא מוגדרת.
float → long	אם הערך שלם, ונמצא בטווח של **long**, לא נאבד דבר. אם הערך בטווח של **long** אבל מכיל שבר, נאבד את השבר: **5,731.78 → 5,731** אם הערך מחוץ לטווח, התוצאה לא מוגדרת.
long → float	כיוון של-**float** רק 6 ספרות דיוק, נקבל את הערך של ה-**long** אבל **במקורב**, כלומר אנו נאבד דיוק: **760,553,411,987 → 7.60553e+11**

דוגמאות ל-Casting

```
int cake_num = 5, children = 3;
double cake_per_child;
```

כמות העוגה שיש לכל ילד:

```
cake_per_child =
  (double)cake_num /
  children ;
```

```
cake_per_child =
  cake_num /
  (double)children ;
```

```
cake_per_child =
  (double)(cake_num / children);
```

מה תהיה התוצאה?

דוגמאות ל-Casting

```
short num1, num2;
long product;
product = num1 * num2;
```

```
product = (long)num1 * num2;
```

```
product = num1 * (long)num2;
```


אופרטורים

- אופרטור הוא פעולה של C, המקבלת ערך יחיד או זוג ערכים, ומחזירה ערך כלשהו.
- אופרטור אונארי (unary operator) מקבל ערך יחיד.
- אופרטור בינארי (binary operator) מקבל זוג ערכים.
- ישנם מספר אופרטורים המשנים את הערכים שהם מקבלים. השפעה זו נקראת תוצאת לוואי (side-effect) של האופרטור.

אופרטורים אונאריים

- מינוס מתמטי: `-a`
- casting: `(long)num`
- אופרטורי קידום:

`a++`	`a--`	`++a`	`--a`

מחזירים את הערך המקורי של **a** ומקדמים אותו ב-1 (או -1) (הקידום נעשה **לאחר** ההחזרה)

מקדמים את **a** ב-1 (או -1), ומחזירים את הערך החדש של **a** (הקידום נעשה **לפני** ההחזרה)

אופרטורים בינאריים

- פעולות חשבון:

a+b	a-b	a*b	a/b	a%b

- השוואות:

a==b	a!=b	a<b	a>b	a<=b	a>=b

- פעולות לוגיות:

a && b	a || b

- השמות:

a=b	a+=b	a-=b	a*=b	a/=b

דוגמאות

```
int x = 5, y ;

y = -x + x ;
y = ++x ;
y = x++ ;

x = ++x + x++ ;
```

x = 5 , y = 0

x = 6 , y = 6

x = 7 , y = 6

אל תנסו את זה בבית!

סכום סדרה הנדסית

```
double a1 = 5, q = 3.7;
double sum;
int i = 1, n = 100;

sum = a1;
while (i < n) {
  a1 = a1 * q;
  sum = sum + a1;
  i++;
}
```

סדרה הנדסית:

- האיבר הראשון הוא a_1
- כל איבר נוסף מוכפל ב-q
- סכום הסדרה הוא:

$$a_1 + a_1 q + a_1 q^2 + \ldots$$

טבלת הקדימויות

שפת C מגדירה טבלת קדימויות הקובעת באיזה סדר מופעלים האופרטורים השונים. קדימויות שכדאי לזכור הן:

++ -- (casting)
* / %
+ -
= += -= *= /=

ניתן למצוא את טבלת הקדימויות המלאה בספרי C ובאינטרנט.

מה לגבי רצף אופרטורים עם אותה הקדימות?

- כאשר הקומפיילר רואה שרשרת פעולות שכולן עם אותה הקדימות, הוא צריך להחליט באיזה סדר לבצע אותן.
- הוא יכול לבצע אותן משמאל לימין או מימין לשמאל.
- שפת C מגדירה לכל קבוצת אופרטורים בעלת אותה קדימות את הכיוון בו יש לבצע שרשרת של אופרטורים כאלה.
- תכונה זו נקראת **אסוציאטיביות**.
- האסוציאטיביות של כל האופרטורים מופיעה בטבלת הקדימויות.

טבלת הקדימויות (2)

הנה טבלת הקדימויות המעודכנת, הכוללת גם את האסוציאטיביות של כל קבוצת אופרטורים:

`++ -- (casting)`	
`* / %`	שמאל לימין
`+ -`	שמאל לימין
`= += -= *= /=`	ימין לשמאל

דוגמאות לאסוציאטיביות

פעולות חשבון (שמאל לימין):

```
10 + 20 - 5 - 2 = 23
10 * 20 / 8 / 5 = 5
```

השמות (ימין לשמאל):

```
x = 4; y = 5;
z = y += x *= 5;
```

→

```
x == 20
y == 25
z == 25
```

פעולת השמה מחזירה את הערך שנכתב למשתנה

ושוב: סכום סדרה הנדסית

```
double a1 = 5, q = 3.7;
double sum;
int i = 1, n = 100;

sum = a1;
while (i++ < n) {
  sum += a1 *= q;
}
```

בהשוואה נלקח בחשבון הערך של `i` לפני הקידום

משנה את הערך של `a1` ומחזיר את ערכו החדש