פרק 5
ביטויים לוגיים
ומשפטי תנאי

ערכי אמת

- מבחינים בלוגיקה בין שני ערכי אמת: true ו- false.
- בשפת C אין טיפוס המיועד להחזקת ערכי אמת.
- לצורך ייצוג ערכי אמת משתמשים בטיפוס **int**.
- הערך 0 מייצג false (שקר).
- כל ערך אחר מייצג true (אמת).

`0`	false
`1, 5, -200, 'a'`	true

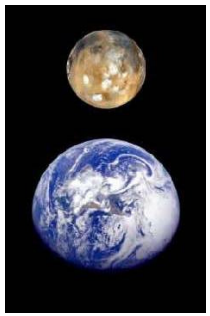
דוגמאות

```
if (0) {
  printf("this will never be printed");
}
if (-30) {
  printf("this will always be printed");
}
```

```
int n;
scanf("%d", &n);
if (n) {
   printf("n is non-zero!");
}
```

אופרטורי השוואה

- אופרטורים בינאריים המחזירים ערך מטיפוס **int**.
- אם ההשוואה נכונה האופרטור מחזיר 1, אחרת הוא מחזיר 0.



A שווה ל-B	`A == B`
A שונה מ-B	`A != B`
A גדול מ-B	`A >  B`
A קטן מ-B	`A <  B`
A גדול או שווה ל-B	`A >= B`
A קטן או שווה ל-B	`A <= B`

אופרטורי השוואה

- האסוציאטיביות של כל אופרטורי ההשוואה היא משמאל לימין.
- לא כדאי לשרשר השוואות... מה תדפיס התוכנית הבאה?

```
int x = 2;
if ( 3 < x < 5 ) {
   printf("oops...");
} else {
   printf("great!");
}
```


אופרטורי השוואה

- שימו לב ששני הביטויים הבאים שקולים כיוון ש-(<) מחושב משמאל לימין :

```
3 < x < 5   ⟷   (3 < x) < 5
```

- `(3 < x)` מחזיר 0, ואפס קטן מ-5 ...
- לכן הביטוי מחזיר בסוף 1.
- למעשה, הביטוי הזה תמיד מחזיר 1 ! (מדוע?)

אופרטורים לוגיים

- אופרטורים לוגיים מתייחסים לפרמטרים שלהם כאל ערכי אמת, ומחזירים 1 או 0 על פי הטבלה הבאה.
- הסימונים A ו-B בטבלה מייצגים ביטויים כלשהם.
- האופרטורים הלוגיים מחזירים ערך מטיפוס `int`.

האופרטור	שם הפעולה	מה מחזיר ?
`!A`	not A	היפוך ערך האמת של A (מחזיר 0 אם A מתקיים, ו-1 אם אינו מתקיים).
`A && B`	A and B	מחזיר 1 אם גם A וגם B מתקיימים.
`A || B`	A or B	מחזיר 1 אם לפחות אחד מבין A, B מתקיים.

טבלאות אמת

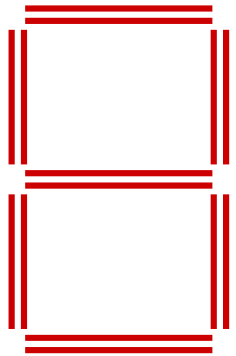
טבלאות אלה מציינות לכל סט פרמטרים אפשרי את ערך הביטוי הלוגי:

A	B	A && B	A || B	!A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

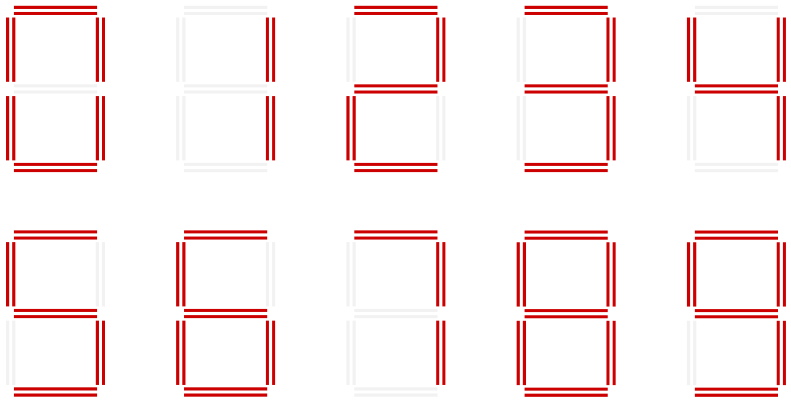
אופרטורים לוגיים

הביטוי	תוצאה
`!5`	0
`!!5`	1
`5 && 0`	0
`5 && 'A'`	1
`5 || (-5)`	1
`5 || 0`	1

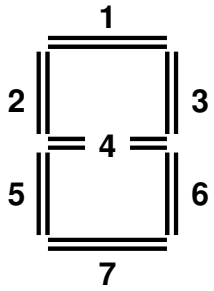
דוגמה: תכנות LED



תכנות LED



תכנות LED

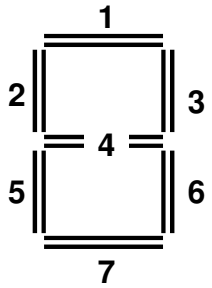


האלגוריתם:

1. קרא ספרה מהמשתמש
2. קבע אילו LED-ים צריכים לדלוק
3. צייר את ה-LED למסך

אילו LED-ים צריכים לדלוק?

```
int led1, led2, led3, led4,
    led5, led6, led7 ;
```




```
led6 = (digit != 2);
```

```
led5 = (digit == 0) || (digit == 2) ||
       (digit == 6) || (digit == 8);
```

```
led2 = (digit == 0) || ((digit > 3) && (digit != 7));
```


החישוב המלא

```
led1 = (digit != 1) && (digit != 4);

led2 = (digit == 0) ||
       ((digit > 3) && (digit! = 7));

led3 = (digit != 5) && (digit != 6);

led4 = (digit != 0) && (digit != 1) &&
       (digit != 7);

led5 = (digit == 0) || (digit == 2) ||
       (digit == 6) || (digit == 8);

led6 = (digit != 2);

led7 = (digit != 1) && (digit != 4) &&
       (digit != 7);
```


Short-Circuit Evaluation

בשפת C החישוב של ביטוי לוגי נעצר כאשר ברור מה ערכו ואין טעם לבדוק את יתר התנאים. לדוגמה :

```
(hair == black) && (eyes == green)
```

אם **hair** אינו **black**, מוחזר 0 והתנאי השני כלל לא נבדק.

```
(speed > 90) || (speed < 55)
```

אם **speed>90**, מוחזר 1 בלא לבדוק את התנאי השני.

Short-Circuit Evaluation

מנגנון החישוב הלוגי המקוצר מתייחס לאופרטורים **&&** ו-**||**.
ישנם שני מקרים בהם החישוב ייעצר מוקדם:

A && B

אם **A** לא מתקיים - ברור שהתוצאה היא 0 ולכן החישוב יעצור.

A || B

אם **A** מתקיים - ברור שהתוצאה היא 1 ולכן החישוב יעצור.

דוגמאות

```
led4 = (digit != 0) && (digit != 1) &&
       (digit != 7);
```

- אם **digit==0** אז התנאי הראשון מתאפס. כעת תוצאת ה-**&&** היא בוודאות 0 ולכן החישוב נעצר מייד ומחזיר 0 ללא ביצוע שתי ההשוואות הבאות.
- אם **digit==1** אז ההשוואה הראשונה מתבצעת בכל מקרה, אבל התנאי השני מתאפס, ולכן אחריו כבר ברור שתוצאת ה-**&&** היא 0. בשלב זה החישוב נעצר מייד, ולא מתבצעת ההשוואה האחרונה.

דוגמאות

```
led2 = (digit == 0) ||
       ((digit > 3) && (digit != 7));
```

- אם **digit==0** אז התנאי הראשון מתקיים. בשלב זה ברור שתוצאת ה-**||** היא 1 ולכן החישוב נעצר מייד ומחזיר 1.
- אם **digit==1** אז ההשוואה הראשונה מתבצעת בכל מקרה, אבל התנאי **digit>3** (ששייך לאופרטור **&&**) אינו מתקיים, ולכן אין צורך להמשיך בחישוב ה-**&&**. ה-**&&** מחזיר 0 בלא שההשוואה האחרונה התבצעה, ובסה"כ נקבל **led2=0**.

שימוש בלוגיקה מקוצרת

- תכונת הלוגיקה המקוצרת מאפשרת לשלב כמה בדיקות בו זמנית, גם כאשר בדיקה אחת תלויה בתוצאה של בדיקה קודמת:

```
if ((x != 0) && (1/x > 12))
```

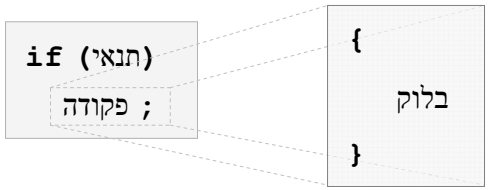
- בדוגמה הבאה רוצים לקרוא מילה שאורכה לכל היותר **maxlen** תווים. יש לעצור את הלולאה כאשר עוברים את האורך המותר, או לחילופין כשהקלט מסתיים. שימו לב שבמידה ומגיעים לאורך המילה המקסימאלי, <u>איננו</u> רוצים לקרוא תו נוסף.

```
if (wordlen>maxlen || (c=getchar())==EOF)
```

בקוד זה, אם **wordlen>maxlen** אזי ה-**if** מיד מתבצע, והקריאה ל-**getchar()** אינה מתרחשת.

פקודת if

- **if** מגדירה קטע קוד שביצועו מותנה בקיום תנאי מסוים.
- התנאי הוא ביטוי כלשהו שניתן לייחס לו ערך אמת.
- חישוב ערכו של התנאי מתבצע (כמו כל חישוב לוגי) בעזרת לוגיקה מקוצרת.
- אם (ורק אם) התנאי מתקיים, נבצע את הפקודה המתאימה.
- ניתן לציין פקודה יחידה או לתת **בלוק** של פקודות.



פקודת if-else

- אם תנאי ה-**if** לא מתקיים, ניתן להגדיר קטע קוד חלופי לביצוע באמצעות **else**.
- את פקודת ה-**else** והקוד השייך לה יש למקם מייד לאחר קטע הקוד השייך ל-**if**. במקרה של אי-קיום תנאי ה-**if**, יתבצעו במקום זאת הפקודה (או הבלוק) של ה-**else**.
- ה-**else** נחשב כחלק מפקודת ה-**if**, ואין לו קיום כפקודה עצמאית. לכן, בכל מקום שבו יש לכתוב פקודה יחידה, ניתן לכתוב גם זוג **if-else**.

```
if (תנאי)
   פקודה 1 ;
else
   פקודה 2 ;
```


דוגמאות

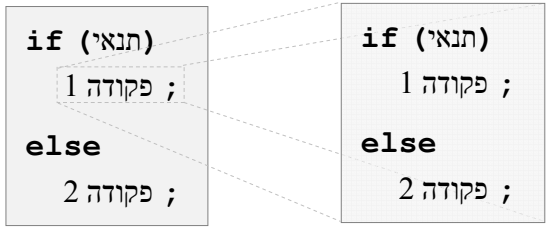
```
if ( (c=getchar()) != EOF ) {
   printf("read was successful");
} else {
   printf("end-of-file reached");
}
```

- ומה קורה כאן ??

```
if (a++)
   printf("a was not zero");
else
   printf("a is not zero");
```


שרשור if-else

- הפקודה השייכת ל-**if** (כמו גם הפקודה השייכת ל-**else**) יכולה להיות בעצמה פקודת **if**... או אפילו זוג **if-else** (כיוון שהם ביחד נחשבים פקודה יחידה).
- מכאן נובע הכלל: **else תמיד שייך ל-if הקרוב ביותר**.



דוגמאות

- בקוד הבא קוראים קלט רק במידה והמשתנה **get_input** מציין שיש לעשות כן. שימו לב שה-**else** שייך ל-**if** השני.

✗

```
if (get_input==1)
  if (scanf("%d",&input) == 1)
    printf("success");
  else printf("error");
```

- ככלל, עדיף בכל זאת להשתמש בתווי { } כדי לעשות את הקוד ברור:

✓

```
if (get_input==1) {
  if (scanf("%d",&input) == 1)
    printf("success");
  else printf("error");
}
```

דוגמאות

```
if (output_ready)
  if (printf("%d",output) < 1) return 0;
else printf("no output for you!");
```

שימו לב: הקוד שלמעלה איננו מבצע את מה שהתכוונו!
למרות צורת הכתיבה המבלבלת, ה-**else** שייך ל-**if** השני ולא לראשון. כדי לגרום לשיוך הנכון צריך להוסיף **{}** כך:

```
if (output_ready) {
  if (printf("%d",output) < 1) return 0;
}
else printf("no output for you!");
```


if-else שרשור

- רצף של **if-else** משורשרים משמש לבדיקה של מספר אופציות בזו אחר זו. נהוג לכתוב את האוסף כולו במבנה "elseif".
- שימו לב שצורת הכתיבה של הרצף **שונה** מהצורה המקובלת. אילו היינו כותבים את הרצף כמקובל, היינו צריכים לזוז למטה וימינה בכל **if** נוסף, כיוון שהוא למעשה משמש בתור הפקודה השייכת ל-**else** שלפניו.

```
if (…) {
   …
}
else if (…) {
   …
}
else if (…) {
   …
}
else {
   …
}
```


if-else שרשור

```
if (a<0) {
   …
}
else if (a==0) {
   …
}
else if (a>0) {
   …
}
else {
   printf("???");
}
```

מבנה else-if

כתיבה רגילה

```
if (a<0) {
   …
}
else
   if (a==0) {
      …
   }
   else
      if (a>0) {
         …
      }
      else {
         printf("???");
      }
```


קריאת הספרה מהמשתמש

- בחזרה לתוכנית שלנו: עלינו לקרוא ספרה מהמשתמש.
- יש לבדוק שלא הייתה בעיה בקריאה ל-**scanf()**, וכן לוודא שהמספר שקראנו הוא אכן ספרה חוקית.

```
int digit ;

if( scanf("%d", &digit) < 1 ) {
   printf("error in scanf\n"); return 0;
}

if (digit<0 || digit>9) {
   printf("invalid digit\n"); return 0;
}
```

קריאת הספרה מהמשתמש

- ניתן גם לכתוב את כל הבדיקות בפקודת **if** אחת!

```
if ( (scanf("%d", &digit) < 1) ||
     (digit<0 || digit>9) )
{
   printf("error reading digit\n");
   return 0;
}
```

- בזכות תכונת הלוגיקה המקוצרת, במקרה שה-**scanf()** נכשל התוכנית מפסיקה מיד את חישוב תנאי ה-**if** (ללא בדיקת הטווח של **digit**), ועוברת לביצוע תוכן ה-**if**.

תוכנית ה-LED עד כה

- קראנו את הספרה וחישבנו אילו LED-ים דולקים. כעת נותר רק לצייר אותם למסך !

```
int digit;

if ((scanf("%d", &digit) < 1) ||
    (digit<0 || digit>9) )
{
   printf("error reading digit\n");
   return 0;
}

led1 = (digit != 1) && (digit !=4 );

led2 = …
```


ציור LED-ים למסך

- נצייר כל נורה אופקית כרצף של ארבעה מקפים (`----`)
- נצייר כל נורה אנכית כרצף של ארבעה קוים אנכיים (`'|'`)
- נתחיל ב-**led1**, זו הנורה האופקית העליונה.
  הפקודה המתאימה היא:

```
if (led1) {
   printf("----\n");
} else {
   printf("\n");
}
```

... ואולי אפשר לקצר את זה לשורה אחת ?

אופרטורים טרינאריים (ternary)

- אופרטורים המקבלים שלושה פרמטרים
- יש רק אחד כזה – האופרטור **? :**

```
(תנאי) ? (ביטוי 1) : (ביטוי 2)
```

- אם התנאי מתקיים, האופרטור מחזיר את ערכו של ביטוי 1, אחרת הוא מחזיר את ערכו של ביטוי 2.
- לדוגמה, הקוד הבא הופך אות כלשהי לאות גדולה:

```
c = (c>='a' && c<='z') ? (c+'A'-'a') : c;
```

האופרטור ? :

- התנאי של האופרטור יכול להיות תנאי מורכב, והוא מחושב בהתאם לכללי החישוב הרגילים (לוגיקה מקוצרת).
- הביטויים 1 ו-2 יכולים להיות מכל טיפוס שהוא.
- אם הטיפוסים של הביטויים 1 ו-2 שונים, תבוצע המרה אוטומטית של אחד מהם לטיפוס של השני, לפי אותם הכללים שראינו עבור אופרטורים בינאריים.
- למשל, שימוש נפוץ של האופרטור הוא פעולת מקסימום:

```
max = (a>b) ? a : b ;
```

⟷

```
if (a>b)
   max = a;
else
   max = b;
```


הקוד עבור LED1

- הביטויים של האופרטור ? : יכולים להיות גם מחרוזות.
- זה מאפשר לנו לקצר את הקוד עבור **led1** לשורה יחידה:

```
printf("%s\n", led1? "----" : "");
```

- (תזכורת: תג הבקרה **%s** מציין מחרוזת בפקודות קלט/פלט)
- אותו הקוד ישמש אותנו גם לציור יתר הנורות האופקיות (**led4** ו-**led7**).

הקוד עבור LED2/3

- נשתמש עתה באופרטור ? : עם פרמטרים שהם תווים.
- זכרו שהתג **%c** מסמן תו בודד בפקודות קלט/פלט.
- הקוד הבא מציג על פני ארבע שורות את **LED2** ו-**LED3** (מדוע זה עובד?)

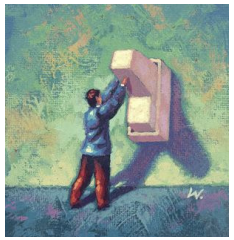
```
i=0;
while (i < 4)
{
   printf("%c    %c\n",
       led2 ? '|' : ' ' ,
       led3 ? '|' : ' ' );
   i++;
}
```


סיכום: הקוד המלא לציור הLED-ים

```
printf("%s\n", led1 ? " ---- " : "");

i = 0; while (i++ < 4) {
   printf("%c    %c\n",
      led2 ? '|':' ', led3 ? '|':' ');
}
printf("%s\n", led4 ? " ---- " : "");

i = 0; while (i++ < 4) {
   printf("%c    %c\n",
      led5 ? '|':' ', led6 ? '|':' ');
}
printf("%s\n", led7 ? " ---- " : "");
```

הפקודה switch

```
switch (expression)
{
   case value_1 : פקודה 1
                  פקודה 2
                  …
                  break;

   case value_2 : פקודה 1
                  …
                  break;
   …
   case value_n : ( … )
   default      : ( … )
}
```



- **switch** מיועדת למקרה שבו נתון לנו ביטוי כלשהו, ועלינו לבחור בין פעולות שונות על פי ערכו.
- מבנה הפקודה **switch** :

פעולת הפקודה switch

- ראשית, הערך של הביטוי *expression* מחושב. ביטוי זה חייב להיות מטיפוס שלם כלשהו.
- בהתאם לערך שמתקבל, התוכנית קופצת לשורת ה-**case** בה מופיע ערך זה (ערכי ה-**case** עצמם יכולים להיות ביטויים קבועים בלבד הידועים בזמן הקומפילציה, ולא משתנים או תוצאות חישוב, למשל).
- כעת התוכנית מבצעת את כל הפקודות הרשומות בזו אחר זו, עד שהיא מגיעה לפקודה **break** שגורמת ליציאה מה-**switch**.
- אם אין אף שורת **case** מתאימה, התוכנית קופצת לשורה שבה כתוב **default**.
- שורת ה-**default** היא אופציונאלית. אם התוכנית לא מוצאת שורת **case** מתאימה, וגם אין שורת **default**, אז היא יוצאת מה-**switch** בלי לעשות דבר.

דוגמה: מחשבון פשוט

```
int a, b, result; char op;                    Run!

scanf("%d %c %d", &a, &op, &b);

switch (op) {
  case '+': result = a+b; break;
  case '-': result = a-b; break;
  case '*': result = a*b; break;
  case '/': result = a/b; break;
  default :
    printf("unknown operation\n");
    return 0;
}

printf("%d%c%d = %d\n", a, op, b, result);
```


הערות לגבי switch

- פקודות ה-**break** בתוך ה-**switch** הן אופציונאליות.
  - אם התוכנית אינה מוצאת **break** בסוף רצף הפקודות של ה-**case**, היא פשוט **ממשיכה** לבצע את הפקודות של ה-**case** הבא, עד שהיא מגיעה ל-**break** כלשהו שמפסיק את ביצוע ה-**switch**.
  - אם התוכנית מגיעה באופן זה לסוף בלוק ה-**switch**, היא יוצאת מהבלוק (ולכן בעיקרון לא צריך **break** בסוף ה-**case** האחרון. אבל זה נחשב תכנות רע להשמיט אותו).
  - מותרות שורות **case** ריקות שאינן מכילות אף פקודה. במקרה זה התוכנית ממשיכה לבצע את הפקודות של ה-**case**-ים הבאים, עד שהיא נתקלת ב-**break** כלשהו.
- בלוק ה-**switch** כולו נחשב כפקודה אחת, ולכן ניתן לשים בלוק **switch** בכל מקום בו נדרשת פקודה יחידה.

דוגמה פשוטה

```
switch (digit) {
   case 0:
   case 1:  printf("undefined!\n");
            break;
   case 2:
   case 3:
   case 5:
   case 7:  printf("prime!\n");
            break;
   case 4:
   case 6:
   case 8:
   case 9:  printf("composite!\n");
            break;
   default: printf("not a digit...\n");
            break;
}
```


דוגמה אמיתית

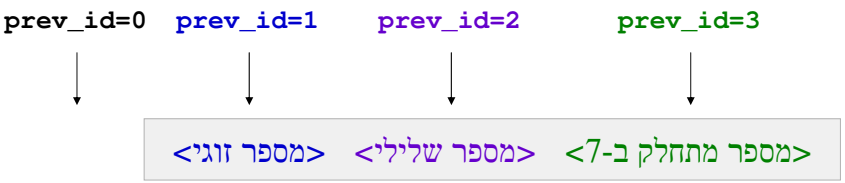
- "אמיתית" – כיוון שתרגיל ברוח זו הופיע בקורס תכנות מתקדם.
- התוכנית מקבלת כקלט בין 0 ל-3 מספרים שלמים.
- המספרים צריכים לקיים את הפורמט הבא:

<מספר מתחלק ב-7> <מספר שלילי> <מספר זוגי>

- כל אחד מן הפרמטרים האלה הוא **אופציונאלי** (זו משמעות ה- < >).
- צריך לוודא שהקלט חוקי, כלומר שהמספרים שקיבלנו מקיימים את התכונות המצוינות **ובאותו הסדר** שהפורמט מכתיב.
- איך היינו מתכנתים את זה עם פקודת `if`? וכיצד היה משתנה בקוד אילו במקום שלושה היינו צריכים לקרוא 4 פרמטרים, או יותר?

דוגמה אמיתית

- נעזר במשתנה אינדקס שייקרא `prev_id`. משתנה זה יכיל את האינדקס של הפרמטר האחרון שקראנו.
- בתחילה, `prev_id` מאותחל ל-0.
- בכל פעם שקוראים מספר נוסף, נבדוק מאיזה טיפוס הוא, על פי סדר הפרמטרים שהגדרנו. ברגע שנמצא התאמה, נדע לאיזה פרמטר המספר שקראנו מתאים, ונקדם את `prev_id` לאינדקס של פרמטר זה.
- נתחיל את החיפוש רק מהטיפוס של הפרמטר הקודם ואילך!



התוכנית לבדיקת הקלט :

```
prev_id = 0;
while ( (prev_id < 3) &&
        (scanf("%d", &num) == 1) )
{
  switch(prev_id)
  {
    case 0: if (num%2 == 0)
                { prev_id = 1; break; }
    case 1: if (num<0)
                { prev_id = 2; break; }
    case 2: if (num%7 == 0)
                { prev_id = 3; break; }
  }
}
```

איך זה עוזר לנו לתוכנית הLED-ים?

- עבור רוב הספרות, יש יותר LED-ים דלוקים מאשר כבויים.
- נשתמש ב-**switch** על מנת לכתוב קוד קצר יותר לחישוב ה-LED-ים הדלוקים.

1. נאתחל את כל הLED-ים להיות דלוקים:

```
led1 = led2 = led3 = led4 =
led5 = led6 = led7 = 1;
```

2. בהתאם לערך של **digit**, נכבה את ה-LED-ים המיותרים (באמצעות **switch**).

הקוד של ה-switch (גירסה 1)

```
switch (digit)
{
  case 0: led4=0; break;
  case 1: led5=led2=led7=led4=led1=0; break;
  case 2: led2=led6=0; break;
  case 3: led5=led2=0; break;
  case 4: led1=led5=led7=0; break;
  case 5: led3=led5=0; break;
  case 6: led3=0; break;
  case 7: led5=led2=led7=led4=0; break;
  case 9: led5=0; break;
}
```


אותו הקוד, בסדר קצת שונה...

```
switch (digit)
{
  case 0: led4=0; break;
  case 2: led2=led6=0; break;
  case 4: led1=led5=led7=0; break;

  case 5: led3=led5=0;  break;
  case 6: led3=0;       break;

  case 1: led5=led2=led7=led4=led1=0; break;
  case 7: led5=led2=led7=led4=0;      break;
  case 3: led5=led2=0;                break;
  case 9: led5=0;                     break;
}
```


הקוד של ה-switch (גירסה 2)

```
switch (digit)
{
  case 0: led4=0; break;
  case 2: led2=led6=0; break;
  case 4: led1=led5=led7=0; break;
  case 5: led5=0;
  case 6: led3=0; break;
  case 1: led1=0;
  case 7: led7=led4=0;
  case 3: led2=0;
  case 9: led5=0; break;
}
```

התוכנית המלאה (בתוספת לולאה)

```
int main()
{
  int digit, i, led1, led2, led3, led4, led5, led6, led7;

  printf("enter a digit: ");
  while ((scanf("%d", &digit)==1 ) && (digit>=0) && (digit<=9))
  {
      led1 = led2 = led3 = led4 = led5 = led6 = led7 = 1;
      switch (digit) {
          case 0: led4 = 0; break;
          case 2: led2 = led6 = 0; break;
          case 4: led1 = led5 =  led7 = 0; break;
          case 5: led5 = 0;
          case 6: led3 = 0; break;
          case 1: led1 = 0;
          case 7: led4 = led7 = 0;
          case 3: led2 = 0;
          case 9: led5 = 0; break;
      }
```

Run!

התוכנית המלאה (בתוספת לולאה)

```
      printf("\n%s\n", led1 ? " ---- " : "");

      i = 0; while (i++ < 4) {
         printf("%c    %c\n", led2 ? '|':' ' , led3 ? '|':' ');
      }

      printf("%s\n", led4 ? " ---- " : "");

      i = 0; while (i++ < 4) {
         printf("%c    %c\n", led5 ? '|':' ' , led6 ? '|':' ');
      }

      printf("%s\n", led7 ? " ---- " : "");

      printf("enter a digit: ");
   }
   return 0;
}
```