גירסה 1.00 – 6.2.2005

מערכים דו ממדיים

ניר אדר

מסמך זה הורד מהאתר http://underwar.livedns.co.il.

אין להפיץ מסמך זה במדיה כלשהי, ללא אישור מפורש מאת המחבר.
מחבר המסמך איננו אחראי לכל נזק, ישיר או עקיף, שיגרם עקב השימוש במידע המופיע במסמך, וכן לנכונות התוכן של הנושאים המופיעים במסמך. עם זאת, המחבר עשה את מירב המאמצים כדי לספק את המידע המדויק והמלא ביותר.

כל הזכויות שמורות ל**ניר אדר**

Nir Adar
Email: underwar@hotmail.com
Home Page: http://underwar.livedns.co.il

אנא שלחו תיקונים והערות אל המחבר.

מערכים דו ממדיים

מבוא ותחביר בסיסי

מערכים חד ממדיים מאפשרים לנו ליצור סדרות של נתונים. סידרה של נתונים, כאשר לכל איבר יש מספר מזהה, שימושית בתוכניות רבות. כמה דוגמאות למערכים חד ממדיים: מערך השומר נתונים של שחקנים שונים במשחק – כל תא שומר מידע על שחקן אחד, מערך השומר מידע לכל יום בחודש (לפי האינדקס) וכו'. נרחיב את רעיון המערכים למספר ממדים. מערכים דו ממדיים מאפשרים לנו לייצג נתונים כגון מטריצות, לוחות משחק וכדומה. לעתים במסמך זה נכנה מערכים דו ממדיים גם בשם **מטריצות**.

התחביר ליצירת מערך דו ממדי:

```
<type> <array_name>[DIM_X][DIM_Y];
```

לדוגמא:

```
int mat[10][10];
```

פקודה זו תיצור מערך בגודל 10 על 10 שכל תא הוא מסוג int.
האינדקס של התא בפינה השמאלית העליונה הוא mat[0][0] ובפינה הימנית התחתונה הוא mat[9][9].
כאשר נכתוב כפי שכתבנו לעיל, כל התאים יכילו בהתחלה נתונים אקראיים.

הכנסת נתונים למערך דו ממדי נעשית על ידי תחביר דומה לזו המשמשת להכנסת נתונים למערך חד ממדי.
התחביר הינו:

```
<array_name>[X][Y] = value;
```

למשל, נשים את הערך 1 בתא (0,0):

```
mat[0][0] = 1;
```

קריאת נתונים ממערך דו ממדי, באופן דומה. למשל, קריאת תוכנו של התא (0, 0) והשמתו למשתנה:

```
int i;
i = mat[0][0];
```

אתחול ערכים למערך דו ממדי יכול לעשות בצורה מפורשת בעת יצירתו, בדומה לאתחול של מערך.
כל הגדרת האתחול מוקפת בסוגריים מסולסלות. כל שורה במערך הדו ממדי מאותחלת בדומה למערך חד ממדי, על ידי איברים המוקפים בסוגריים מסולסלות ומופרדים ביניהם על ידי פסיקים. שורות במערך מופרדות אף הן על ידי פסיקים. נאתחל לדוגמא מטריצה בגודל 5×5 בלוח הכפל:

```
int mat[5][5] = {   { 1,  2,  3,  4,  5 },
                    { 2,  4,  6,  8, 10 },
                    { 3,  6,  9, 12, 15 },
                    { 4,  8, 12, 16, 20 },
                    { 5, 10, 15, 20, 25 }   };
```

בדומה למערכים, אם נאתחל רק חלק מהתאים, שאר התאים יאותחלו להיות 0. לדוגמא, אם נכתוב:

```
int mat[5][5] = {   { 1,  2,  3 },
                    { 2,  4,  6,  8, 10 },
                    { 3,  6,  9, 12, 15 },
                    { 4,  8, 12, 16, 20 } };
```

אז המטריצה תראה בזיכרון כך:

1	2	3	0	0
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
0	0	0	0	0

דוגמא 1 – איפוס אברי המטריצה

אחת הפעולות הנפוצות במטריצות היא איפוס כל האיברים במטריצה (השמה של 0 בכל התאים בה).
מתי נרצה לעשות דבר כזה? למשל במידה ואנחנו מתכנתים משחק, נרצה ליצור לוח משחק ריק (נניח ש-0 מציין בתוכנית שלנו משבצת ריקה בלוח). לאיפוס מטריצה בגודל 10×10, הקוד ייראה כך:

```
int mat[10][10];
int i, j;
for (i = 0; i < 10; i++)
{
	for (j = 0; j < 10; j++)
	{
		mat[i][j] = 0;
	}
}
```

נשים לב בדוגמא כי השתמשנו בלולאה בתוך לולאה. מצב זה קורה לרוב כשאנחנו עובדים עם מטריצות. לולאה אחת עוברת על השורות (כלומר, באיטרציה הראשונה היא מתייחסת לשורה הראשונה, באיטרציה השניה ללולאה השניה וכו'), והלולאה השניה עוברת על האיברים בשורה עצמה, אחד אחרי השני.

דוגמא 2 – לולאה שסוכמת אברי מטריצה

נציג דוגמא נוספת לשימוש במטריצות: לולאה העוברת על כל איברי מטריצה בגודל M על N וסוכמת אותם. גם בדוגמא זו אנחנו משתמשים בלולאה בתוך לולאה, וכן במשתנה סכום ששומר כל הזמן את סכום התאים אותם עברנו. אנו מניחים כי M ו-N הם קבועים שהוגדרו במקום אחר בתוכנית.

```
int mat[M][N];
int sum = 0;
int i, j;
for (i = 0; i < N; i++)
{
	for (j = 0; j < M; j++)
	{
		sum += mat[i][j];
	}
}
printf("%d\n", sum);
```

Off topic – שימוש ב-typedef

נציג כעת את השימוש ב-typedef. נושא זה לא קשור ישירות למערכים דו ממדיים, אולם מיד נשתמש בו, ולכן אציג אותו עבור אלו שלא מכירים אותו.

typedef מאפשר למשתמש להגדיר שם נרדף לסוג משתנה הקיים בשפה. למשל:

```
typedef int length;
```

ביטוי כזה אומר כי מעתה length הוא שם נרדף ל-int.

כעת נוכל להגדיר משתנים מסוג length:

```
length x = 7;
```

למה זה טוב? השימוש ב-typedef נותן לנו שני יתרונות עיקריים:

- **שיפור קריאות התוכנית** – נניח שאנחנו כותבים תוכנית ובה משתנים רבים. אם יהיו לנו 10 משתני int יהיה קשה אולי להבין מה המשמעות של כל אחד מהם – הרי כולם מספרים. אבל אם יהיו לנו 3 משתנים מסוג length, עוד שני משתנים מסוג grade וכדו', הרי שרק מקריאת סוג המשתנה כבר יהיה לנו מושג מה הולך להיות בתוך המשתנה.
- **גמישות לשינויים עתידיים** – הצהרת typedef מופיעה פעם אחת בתוכנית, ולאחר מכן אנחנו מסוגלים להשתמש בסוג החדש שהגדרנו פעמים רבות. typedef נותן לנו את אותו יתרון ששימוש בקבועים נותן לנו. נניח כתבנו תוכנית בה השתמשנו ב-length כפי שהוגדר בדוגמא, וכעת, עם התרחבות התוכנית, האורכים בתוכנית חורגים מגבולות ה-int. אם היינו משתמשים ב-int ללא typedef, היינו צריכים לעבור על כל אחד ממשתני ה-int בתוכנית, ולהחליט אם הוא קשור לנושא האורך, ואם כן להפוך אותו ל-long, לדוגמא. במקרה שהשתמשנו ב-typedef, כל מה שעלינו לעשות זה לשנות את הגדרת ה-typedef כדי שיהיה שם נרדף ל-long ולא ל-int.

נחזור רגע לדוגמא:

```
typedef int length;
```

כיצד נקרא שורה זו?

הצורה האינטואיטיבית להבין typedef, והלא נכונה, היא לקרוא את זה כך: רואים int משמאל, ו-length מימינו, ואז אומרים "int הוא שם נרדף של length". נכון שבדוגמא שהוצגה זה באמת נכון, אבל הצורה הנכונה להבין typedef היא כזו:

1. נמחק לרגע את המילה typedef:

```
int length;
```

2. המשמעות של הביטוי שקיבלנו זה "length הוא משתנה מסוג int".
3. כעת, הוספת typedef אומרת "במקום להתייחס אל length כמשתנה, נתייחס אליו כאל שם נרדף".

מיד נראה למה צורת הסתכלות זו מועילה לנו.

שימוש ב-typedef עם מטריצות

נדגים איך מגדירים typedef עבור מערך דו-ממדי, שגודלו $M \times N$.

```
typedef int mat[M][N];
```

ננסה להבין מה הגדרנו: נמחק את המילה typedef:

```
int mat[M][N];
```

הגדרה זו יוצרת משתנה בשם mat, שהוא מטריצה בגודל $M \times N$. מכאן: כאשר אנחנו מוסיפים את המילה typedef, אז mat הוא שם נרדף למשתנים שהם מטריצה בגודל $M \times N$.

כעת נוכל ליצור משתנה למשל מסוג mat ולהשתמש בו כמטריצה $M \times N$ לכל דבר, לדוגמא:

```
mat m;
m[0][0] = 7;
```

היתרונות בשימוש ב-typedef למטריצות הן חסכון בכתיבה בתוכניות בהן יש לנו מטריצות רבות, וכן הם מקלים על העברת מערכים דו ממדיים לפונקציות, כפי שנראה מיד.

העברת מערכים דו ממדיים לפונקציות

ישנן מספר דרכים, נדגים דרך תבניתית אחת:

נגדיר typedef עבור המערך הדו ממדי שלנו:

```
typedef int mat[M][N];
```

אחרי הגדרה זו, mat הוא שם נרדף למערך בגודל M על N.

פונקציה המקבלת מערך דו ממדי תקבל פשוט את mat. לדוגמא:

```
void my_function(mat m)
{
	/* do something on m */
}
```

המערך מועבר by-ref, כלומר, אם נשמה את המערך הדו ממדי בתוך הפונקציה, אז גם המערך מחוץ לפונקציה בפונקציה הקוראת יושפע.

מערכים דו ממדיים, בדומה למערכים חד ממדיים, אינם יכולים להיות ערך מוחזר של פונקציה. אנחנו לא יכולים ליצור מערך דו ממדי בתוך הפונקציה ולהחזיר אותו. הסיבה לכך היא ש-C דוגלת בגישה שאומרת שאין עלויות נסתרות לפקודות שאנחנו מבצעים. מכיוון שהערך המוחזר של פונקציה מועתק לתוך המשתנים של הפונקציה הקוראת, אז אם היינו מאפשרים להחזיר מערך דו ממדי יכול להיות שפעולת ה-return היתה לוקחת זמן רב (העתקה של מערך גדול ממקום למקום). אי לכך, החליטו מתכנני השפה שמערכים לא ניתנים להחזרה מפונקציות.

נושאים מתקדמים – ייצוג מערכים דו ממדיים בזכרון

כיצד מיוצג מערך דו ממדי בזכרון? מערך חד ממדי מיוצג על ידי רצף תאים עוקבים בזיכרון. בגלל זה כשאנחנו כותבים arr[5] אנחנו יכולים לכתוב בצורה זהה את הביטוי *(arr+5).
מערך דו ממדי מיוצג בצורה דומה. כל שורה נמצאת ברצף בזיכרון, והשורות נמצאות אחת אחרי השניה.

על ידי ציון אינדקס יחיד בשימוש במערך דו ממדי אנחנו מסוגלים לקבל מצביע להתחלת שורה ספציפית. למשל, נגדיר את המטריצה הבאה:

```
int mat[10][10];
```

הביטוי mat[0] הוא משתנה מסוג *int שמצביע לתחילת השורה הראשונה. הביטוי mat[1] הוא מצביע להתחלת השורה השנייה וכו'. ניתן על ידי מצביע זה לעבור על כל איברי המערך:

```
int i;
int *p = mat[0];
for (i = 0; i < 100; ++i)
{
	printf("%d\t", *p);
	p++;
}
```

בדוגמא זו נדפיס את כל איברי המערך, ולא רק את איברי השורה הראשונה, וזה מכיוון שהתאים נמצאים ברצף בזיכרון.