16. Two-dimensional arrays

Herman Kamper

Every array so far has been a list: a row of boxes, one after the other, and one number in square brackets to say which box you want. That covers a lot, but plenty of data does not come as a list. Marks come as a table of students against tests. An image comes as a grid of pixels. Anything you have ever opened in a spreadsheet has rows and columns rather than a single line.

C handles this with a two-dimensional array. If you have started doing matrices in mathematics then this will look extremely familiar: rows down the side, columns across the top, and a value in every cell. Almost all data processing, and essentially all machine learning, stores its data in this shape.

The good news is that a two-dimensional array is a gentle extension of what you already know. There is one rule that is new, and I will get to it, but most of this lecture is the one-dimensional array with a second subscript bolted on. After that, debugging: how to work out what a program is really doing when it refuses to do what you wanted.

Rows and columns

The formal name for this is a multiple-subscripted array: an array that needs more than one number in brackets to identify an element. Two subscripts is by far the common case, so I will say two-dimensional throughout. Here is one called `c`, with four rows and four columns, and the name of every element written into its own box.

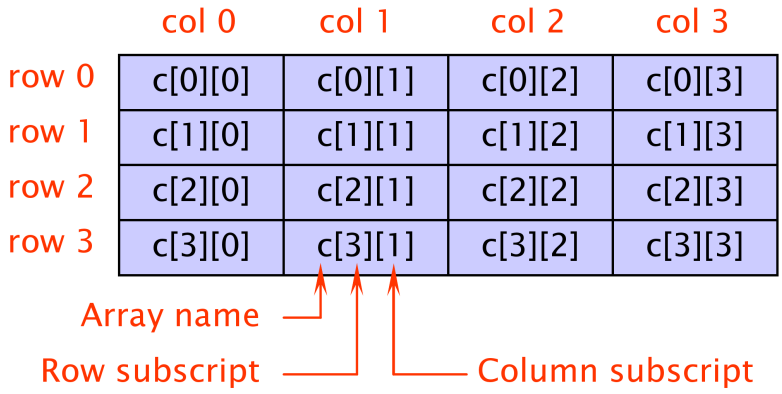

Figure 1: A two-dimensional array `c` with four rows and four columns. The first subscript is the row and the second is the column.

Declaring that array looks like this:

```
int c[4][4];
```

Four rows of four columns is sixteen boxes in total. This line creates them in memory.

To access a particular cell/box, you follow the convention in the figure: the array name comes first, then the row subscript, then the column subscript:

```
c[row][column]
```

Both the rows and columns count from zero, exactly as they did for one-dimensional arrays. So the four rows are numbered 0 to 3 and the four columns are numbered 0 to 3, and `c[2][3]` is the element in row 2, column 3, which in ordinary counting is the third row and the fourth column. It is not row 3, column 2.

Declaring and initialising

You can fill an array in at the moment you declare it, just as you could with a one-dimensional array:

```
int b[2][2] = { {1, 2}, {3, 4} };
```

The initialisers are grouped by row, each row in its own pair of braces. So `{1, 2}` is row 0 and `{3, 4}` is row 1.

C also lets you supply fewer values than there are boxes. Anything you leave out is set to zero:

```
int b[2][2] = { {1}, {3, 4} };
```

Row 0 gets a 1 and then runs out of initialisers, so `b[0][1]` becomes 0. The two declarations produce the arrays below.

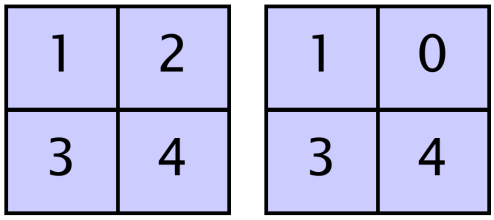

Figure 2: Left: `int b[2][2] = { {1, 2}, {3, 4} };`. Right: `int b[2][2] = { {1}, {3, 4} };`, where the missing initialiser is filled in with zero.

There is one thing here that trips people up constantly. The numbers in square brackets mean two completely different things depending on where they appear. Take a one-dimensional example:

```
int apples[5];
apples[4] = 55;
```

The first line is a declaration, and the 5 is a size: make me five boxes. The second line is not a declaration, and the 4 is an index: go to the box at position 4 and put 55 in it. The `[]`

means different things in different contexts. With two subscripts the same holds, so in `int c[4][4];` the numbers are how many, and in `c[2][3]` they are which one. If you ever find yourself confused about a line involving brackets, the first question to ask is which of the two you are looking at.

An example: student grades

Let us make this concrete. Suppose there are three students in a class, and each of them wrote four exams. That is a table of three rows and four columns, and here it is in C:

```
#define STUDENTS 3
#define EXAMS 4

int studentGrades[STUDENTS][EXAMS] =
{
    {77, 68, 86, 73},
    {96, 87, 89, 78},
    {70, 90, 86, 81}
};
```

Nothing here is new except the layout. I could have written the whole initialiser on one line, but with the rows on separate lines the source code looks like the table it represents, and you can read a student's marks straight off it. Student 0 got 77, 68, 86 and 73; student 1 got 96, 87, 89 and 78; and so on.

What that declaration does is create twelve boxes in memory and put a value in each one. The whole collection goes by the name `studentGrades`, and every individual box is picked out by its row and column, as sketched below.

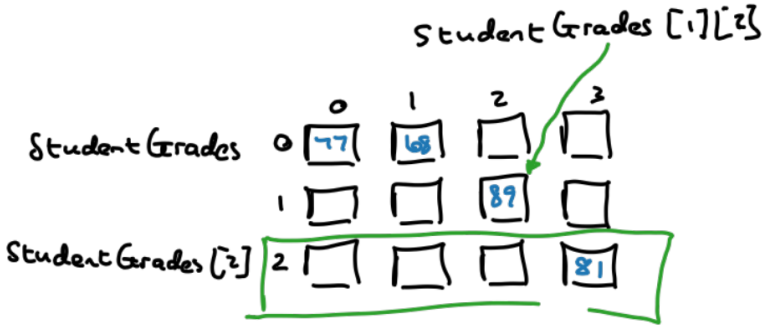

Figure 3: The `studentGrades` array as twelve boxes. `studentGrades[1][2]` picks out a single box holding 89, while `studentGrades[2]` on its own refers to the whole of row 2.

So what does this print?

```
printf("%d\n", studentGrades[1][2]);
```

The tempting answer is 68, and it is wrong. Count from zero. Row 1 is the second row, `{96,`

`87, 89, 78}`, and column 2 is the third entry in it. The answer is 89, which is the box marked in the sketch.

The other annotation is worth noticing now, because we will use it later. `studentGrades[2]`, with one subscript and nothing else, does not refer to a single box. It refers to the whole of row 2, which is itself just an ordinary one-dimensional array of four integers. (This is drawn in green on the figure.)

Printing the whole array

Now two functions. The first prints the entire table; the second computes one student's average. Start with printing, because it is where the nested loop comes in.

To visit every box you need two loops: an outer one that steps through the rows and an inner one that, for each row, steps through the columns.

```
void printArray(const int grades[][EXAMS], int noStudents, int noExams)
{
    int row, column;

    for (row = 0; row < noStudents; row++)
    {
        printf("Grades for student %d: ", row);

        for (column = 0; column < noExams; column++)
        {
            printf("%d ", grades[row][column]);
        }

        printf("\n");
    }
}
```

Read the inner `printf` carefully, because this is the part people misread. It does not print a row. It prints exactly one number, followed by a space, and no newline. It is the loop around it that turns that into a row.

So trace it. `row` is created and set to 0. Zero is less than 3, so we go into the body. We print the label. Then `column` is created and set to 0, and we print `grades[0][0]`, which is 77, and a space. End of the inner loop body, so control goes back to `column++`, not to the outer loop. Now `column` is 1, still less than 4, and we print 68. Then 86, then 73. Then `column` becomes 4, the condition `column < noExams` is false, and only now do we leave the inner loop and reach the `printf("\n")`. That newline ends the line. Then, and only then, control returns to `row++`, `row` becomes 1, and the whole inner loop runs again from scratch for the second student.

There is no magic in any of this. A `for` loop always does the same thing: run the body to the end, go back to the increment, test the condition, and either go round again or fall out the bottom. Nesting one inside another changes nothing about how either of them works. If the order of the

output ever surprises you, the fix is not to memorise the pattern; it is to write down on paper what `row` and `column` hold at each step and what gets printed.

Running it gives what you would hope for:

```
Grades for student 0: 77 68 86 73
Grades for student 1: 96 87 89 78
Grades for student 2: 70 90 86 81
```

Just to make sure there is no confusion: C knows nothing about rows and columns. We just decided to call the variables that; we could have used `i` and `j` and the code would still work, because C is just running through a for loop.

Passing a two-dimensional array to a function

Look again at the first parameter of `printArray`:

```
void printArray(const int grades[][EXAMS], int noStudents, int noExams)
```

Three things are going on there, and the third one is the new rule of this lecture.

First, the array is passed by reference. When you write `printArray(studentGrades, STUDENTS, EXAMS);`, the values 3 and 4 are copied into fresh boxes called `noStudents` and `noExams` that belong to the function and vanish when it returns. The array is not copied. What gets passed is the address where the array starts, so `grades` inside the function refers to the same twelve boxes as `studentGrades` does outside it. This is the call-by-reference behaviour from lecture 14, and it is worth being clear that it is the only reason a function can usefully modify an array.

Second, `const`. It says this function is not allowed to change the array, and the compiler enforces it. Since the array is passed by reference, a stray assignment inside the function could silently corrupt the caller's data, and `const` is how you make that impossible for a function that only needs to read the contents of an array.

Third, and this is the bit that is different: `grades[][EXAMS]`. For a one-dimensional array you write `int b[]` and leave the brackets empty, because the function does not need to know how long the array is. For a two-dimensional array you may leave the first size out, but you must supply the second. You have to tell the function how many columns there are.

Why C needs the number of columns (advanced)

The reason is that a two-dimensional array is not really two-dimensional. In memory there is no grid. There is one straight line of twelve boxes, with row 0 first, then row 1 immediately after it, then row 2:

```
77 68 86 73 96 87 89 78 70 90 86 81
```

The rows-and-columns picture is something the compiler puts on top of that line for your benefit. When you write `grades[row][column]`, the compiler turns it into arithmetic: skip `row` whole rows, then step forward `column` boxes. To skip a whole row it has to know how long a row

is, and that is the number of columns. If it knows there are 4 columns, then `grades[2][1]` becomes "step forward $2 \times 4 + 1 = 9$ boxes from the start", which lands on 90. Leave the number of columns out and the arithmetic is simply not possible, so the compiler refuses.

It would be easy to conclude from this that C is being difficult. It is not, particularly. Every language has to do this same arithmetic somewhere; most of them just keep the array's shape in a hidden field so that you never have to think about it. C's habit is to show you the machinery rather than tidy it away, which is annoying in the moment, but quite useful in the long run, because you end up knowing what the machine is actually doing.

One row at a time

The second function computes the average mark for one student. Now recall what `studentGrades[2]` means on its own: the whole of row 2, an ordinary one-dimensional array of four integers. So the averaging function does not need to know anything about two-dimensional arrays at all.

```
double average(const int studentGrades[], int noExams)
{
    int i;
    double total = 0;

    for (i = 0; i < noExams; i++)
    {
        total += studentGrades[i];
    }

    return total/noExams;
}
```

The parameter is a plain `int[]`, empty brackets and all, because from in here it really is just a list. Note that `total` is a `double` rather than an `int`. It is holding a running sum of whole marks, so an `int` would work for the addition, but making it a `double` means the division at the end is floating-point division and you get 81.75 rather than 81. Also note `total += studentGrades[i];`, which is the shorthand for `total = total + studentGrades[i];` and means exactly the same thing.

Calling it on the third student:

```
printf("\nAverage for student 2: %.2f%%\n", average(studentGrades[2], EXAMS));
```

gives 81.75%, which is 70, 90, 86 and 81 added up and divided by four. Two small formatting points in that line: `%.2f` prints a floating-point number to two decimal places, and to get a literal percent sign out of `printf` you have to write two of them, because a single `%` means a conversion is coming.

The whole program

```c
/*
 * Example illustrating 2D arrays.
 * Author: Herman Kamper
 */

#include <stdio.h>

#define STUDENTS 3
#define EXAMS 4

void printArray(const int grades[][EXAMS], int noStudents, int noExams);
double average(const int studentGrades[], int noExams);

int main()
{
    int studentGrades[STUDENTS][EXAMS] =
    {
        {77, 68, 86, 73},
        {96, 87, 89, 78},
        {70, 90, 86, 81}
    };

    printf("%d\n", studentGrades[1][2]);

    printArray(studentGrades, STUDENTS, EXAMS);

    printf("\nAverage for student 2: %.2f%%\n", average(studentGrades[2], EXAMS));

    return 0;
}

void printArray(const int grades[][EXAMS], int noStudents, int noExams)
{
    int row, column;

    for (row = 0; row < noStudents; row++)
    {
        printf("Grades for student %d: ", row);

        for (column = 0; column < noExams; column++)
        {
            printf("%d ", grades[row][column]);
        }
```

```
        printf("\n");
    }
}

double average(const int studentGrades[], int noExams)
{
    int i;
    double total = 0;

    for (i = 0; i < noExams; i++)
    {
        total += studentGrades[i];
    }

    return total/noExams;
}
```

and its output:

```
89
Grades for student 0: 77 68 86 73
Grades for student 1: 96 87 89 78
Grades for student 2: 70 90 86 81

Average for student 2: 81.75%
```

One thing to notice about the two `#define` constants. `EXAMS` appears five times: in the declaration of the array, in the prototype and the definition of `printArray`, and in both function calls. If the number of exams changed you would still edit one line. That is the entire argument for symbolic constants, and it gets stronger the bigger the program gets.

An array recap

Two-dimensional arrays finish off the arrays material, so this is a reasonable moment to gather the whole lot into one place. There were four ideas:

1. Declaring and using arrays, both one- and two-dimensional, from lecture 13 and this one. The thing to be certain of is the difference between a size and an index, as above.
2. Using arrays to process strings, from lecture 14. A string in C is nothing more exotic than an array of characters, with one addition: a `'\0'` at the end to mark where it stops. Writing `"apples"` in double quotes creates such an array and puts the `'\0'` there for you, which is why a string literal takes up one more box than it has letters.
3. Searching and sorting arrays, from lectures 14 and 15. Linear search, binary search and bubble sort are all just loops over an array, and what makes them worth studying is not the C but the algorithms.

4. Passing arrays to functions, from lecture 14 onwards. An array argument is an address rather than a copy, so a function can change the caller's array. That is call-by-reference. It is the one place where arrays behave unlike every other kind of variable, and if it feels unfamiliar it is the thing to go back to.

Debugging

The last topic is debugging, which is not related only to arrays, but about finding mistakes in general. It is the systematic business of finding out why a program does not do what you meant. You have in fact been doing it since your first program that did not compile, so rather than a treatment of the subject, here are a few tools that make it less painful.

The main one, and the one I use constantly, is `printf`. Put it wherever you are unsure. Print out a variable to see whether it holds what you think it does, and print it inside the loop rather than after it so you can watch it change:

```
printf("row: %d, column: %d, value: %d\n", row, column, grades[row][column]);
```

A `printf` also tells you whether a line runs at all. If the message never appears, execution never got there, and that on its own often locates the problem.

There is a neater version of this that uses two odd little names built into the compiler:

```
printf("%s:%d\n", __FILE__, __LINE__);
```

`__FILE__` expands to the name of the source file and `__LINE__` to the line number, both filled in as the program is compiled. So this line prints out its own position in your source code, which means you can paste the same line in twenty places and still know which one produced which output. Once a program is spread over several `.c` files this becomes quite useful.

The second tool is `exit`:

```
#include <stdlib.h>

exit(-1);
```

It stops the program dead, right there. Not the loop, not the function: the program. Put it after a `printf` and you get a snapshot of the state at a chosen moment without any of the later output scrolling past. Note that this is not what `break` does. `break` jumps out of the enclosing loop and the program carries on afterwards, so in our example a `break` inside `printArray` would still let the average get printed, while `exit(-1)` in the same place would not. The number you pass is the program's exit status, and for debugging it does not matter what you use.

The third tool is the block comment. If you suspect a whole section of a program, wrap it in `/*` and `*/` so the compiler ignores it:

```
/*
    lots of code you want to switch off for a moment
*/
```

If the problem goes away, it is somewhere in what you commented out. Uncomment a bit of it, run again, and keep narrowing until you find the line. This is slow but it never fails, and it beats staring at the code hoping to spot the bug.

Beyond these there is the debugger built into your editor, which lets you pause a program at a chosen line and inspect every variable without adding a single `printf`. It is the proper tool for the job and it is worth learning. But `printf` and `exit` will get you an extraordinarily long way, and they work everywhere.

Looking forward

That is the end of arrays. Along the way we have used the fact several times that an array variable is really an address, and that passing one to a function passes that address rather than a copy. It has been sitting quietly underneath everything from the moment we passed the first array to a function.

The next lecture makes that idea explicit. Addresses are a kind of value in their own right in C: you can store them in variables, and those variables are called pointers. Almost every peculiar thing arrays do turns out to be pointers wearing a disguise.

Exercises

1. Write out, on paper, what the following prints and in what order:

   ```
   int b[2][3] = { {1, 2, 3}, {4, 5, 6} };
   int row, column;

   for (column = 0; column < 3; column++)
   {
       for (row = 0; row < 2; row++)
       {
           printf("%d ", b[row][column]);
       }
   }
   ```

   Note that the loops are the other way round from the ones in this lecture. Then run it and check.

2. What does each of these declare, and what is in it afterwards?

   ```
   int p[3][2] = { {1, 2}, {3, 4}, {5, 6} };
   int q[3][2] = { {1}, {3, 4} };
   int r[3][2] = {0};
   ```

   Write a nested loop that prints each one as a table and confirm your answers.

3. Add a `minimum` function to the grades program that returns the lowest mark in the whole array. It will need both loops, since it has to look at every element. Then add a `maximum` function that returns the highest.

4. Add a `classAverage` function that returns the average over all twelve marks. Then write a loop in `main` that prints every student's own average, using the existing `average` function.

5. Remove `EXAMS` from the first parameter of `printArray`, so it reads `const int grades[][]`, and try to compile. Write down the error message, and explain in one sentence why the compiler cannot manage without that number.

6. Take the `const` off the `grades` parameter of `printArray` and add a line inside that does `grades[0][0] = 0;`. Print `studentGrades[0][0]` in `main` after the call. Explain the result. Then put the `const` back and try the same thing again.

7. Write a program that stores a 3×3 array of integers and prints its transpose: the array with rows and columns swapped, so that the element at row `i`, column `j` of the output is the element at row `j`, column `i` of the input. Do it first by printing in a different order, then again by filling in a second array.

8. Take a program of yours that already works and put `printf("%s:%d\n", __FILE__, __LINE__);` at the top of `main`, inside a loop, and inside a function. Run it and match each line of output to the line of source that produced it. Then replace one of them with `exit(-1)` and confirm you understand exactly where the program stopped.