14 2-dimensional arrays

We have seen arrays of `int, double, char` etcetera. In C, we can have 2-dimensional arrays, such as

```
double a[3][4];
```

(You can also have `char b[4][5][6][2]` etcetera.)

C treats `a` as an array of arrays. To be precise, `a` is an array of 3 arrays each of size 4.

Most of the time we use 2-dimensional arrays for matrix calculations. From this point of view, `a` is a matrix with 3 rows and each row is has 4 entries: in other words, a 3×4 matrix, 3 rows and 4 columns.

Continuing with the matrix `a`, the expression

```
a[2][1]
```

is a particular double-precision number, that in the third row and second column. (Remember that indexing begins at zero.)

Put another way, `a[2]` is an array of doubles, and `a[2][1]` is the second entry in that array.

For the element in the i-th row and j-th column of an array `a[m][n]`,

```
Given
  (type) a[m][n];

The notation in C for array entries is unlike many
programming languages.  It is

NOT a[i,j]
NOR a(i,j)
but the j-th element in a[i]:
   a[i][j]
```

It is in range if $0 \leq i < m$ and $0 \leq j < n$.

Example. Write a routine to read in a matrix and a vector and multiply them.

A matrix is a 2-dimensional array of `double`, say, and a vector is a compatible 1-dimensional array of `double`. The input will include the dimensions of the arrays as well.

We shall use the following input

```
3 4
1 2 3 4
4 5 6 7
7 8 9 10
4
3 -1 4 -5
```

The matrix dimensions are 3×4, and the vector's is 4.

```
#include <stdio.h>
void multiply( double a[3][4], double b[4], double c[3])
{ int i,j;
  double sum;
  for (i=0; i<3; ++i)
  { sum = 0;
    for (j=0; j<4; ++j)
    { sum += a[i][j] * b[j] ; };
    c[i] = sum;
  }
}

void print_matrix ( double a[3][4] )
{ int i, j;
  for ( i=0; i<3; ++i )
  { for ( j=0; j<4; ++j )
    { printf(" %8.3f", a[i][j]);
    }
        // NOTICE the customised formatting: 3 decimal places
        // padded if necessary to 8 characters.
    printf("\n");
  }
}

void print_3vector ( double b[3] )
{ int j;
    for ( j=0; j<3; ++j )
    { printf(" %8.3f", b[j]); }
    printf("\n");
}

void print_4vector ( double b[4] )
{ int j;
    for ( j=0; j<4; ++j )
    { printf(" %8.3f", b[j]); }
    printf("\n");
}

int main()
{ double a[3][4], b[4], c[4];
  int ell,m,n;

  int i,j;

  scanf ("%d %d", &ell, &m); //height and width of a
```

```
  for (i=0; i<ell; i = i+1)
  for (j=0; j<m; j = j+1)
  {  scanf("%lf", &( a[i][j] )) ; }


  scanf ("%d", &n); //height of b
                        // no check that height of b is width of a.
  for (j=0; j<n; j = j+1)
  {  scanf("%lf", &( b[j] )) ; }

  printf("matrix\n"); print_matrix ( a );
  printf("vector\n"); print_4vector ( b );
  multiply(a,b,c);
  printf("product \n"); print_3vector ( c );
}
input:
3 4
1 2 3 4
4 5 6 7
7 8 9 10
4
3 -1 4 -5
ouptut:
matrix
    1.000    2.000    3.000    4.000
    4.000    5.000    6.000    7.000
    7.000    8.000    9.000   10.000
vector
    3.000   -1.000    4.000   -5.000
product
   -7.000   -4.000   -1.000
```

14.1 Arrays as routine arguments

In passing arguments to the multiply routine

```
void multiply( double a[3][4], double b[4], double c[3])
```

the arguments are *copied* from the main program.

If an argument were 'simple,' such as a single `int`, then no change within the routine would affect the value of the corresponding variable in the main routine. But here the result of the multiplication is passed back to the main program through the argument `c`. ***How?***

The answer is: the *value* of an array is the *address of its first entry.* So what gets copied is an *address*, and therefore the results can be transmitted to the calling routine.

In the jargon of programming languages:

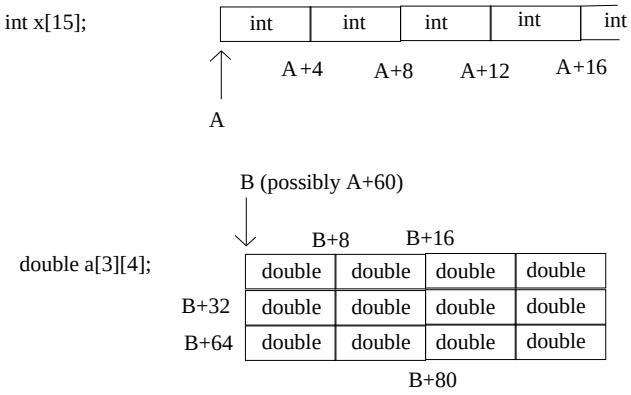

Figure 1: Array-to-memory mappings

- Simple (non-array) arguments are call-by-value,
- Array arguments are *effectively* call-by-reference.

Another example of call-by-reference:

```
scanf("%d", &x)
```

the *address* of `x` is passed, and the value scanned will be stored in `x`.

14.2 Storage mapping functions

Given

```
int x[15];
double a[3][4];
```

the 'value' of `x` is the address of its first element. The arrangement of the array entries in memory is illustrated in Figure 1. The label A is the address of the first entry in `x`. Under the assumption that `a` begins immediately after `x`, B = A+60.

This is because `x` is an array of 15 `int`s, each int is 4 bytes, so the size of the array is 60 bytes. Hence B=A+60 if `a` comes immediately after `x`.

14.3 Address of an entry in a 1-dimensional array

Let

- A = address of first element of `x[i]`, in bytes. Suppose A=1234.
- Let w be the size of each array entry, in bytes: 4 in the above example.
- Let `n` be the size of the array (in ints), 15 in the above example.

- The size of `x` is $4 \times 15 = 60$ in bytes.
- The `i`-th element of `x` has address
$$A + i \times w$$
- For example, given A and w as above, the address of `x[14]` is $1234 + 4 \times 14 = 1290$.
- The address of `x[15]` is 1294. The address is calculated whether or not `i` is in range: 15 is not in range.
- The address of `x[-4]` is 1218.

14.4 Address of an entry in a 2-dimensional array

Suppose `a` is an $m \times n$ array. Let B be the address of its first entry, and let w be the size of each array entry. Given the example

```
double a[3][4];
```

let us again suppose that $B = A + 60 = 1294$.

First give the value of `a[i]`.

Answer. Viewing `a` as an array of m 'rows,' and each row has n entries, the size in bytes of each row is $n \times w$. The value of `a[i]`, which is an address, is

$$B + i \times n \times w$$

This is the address of the first element of the i-th row.

Now the *address* (not the value) of `a[i][j]`: it is

$$C + j \times w$$

where C is the address of the first entry in `a[i]`:

$$B + i \times n \times w + j \times w.$$

For example, the address of `a[2][1]` is

$$1294 + 2 \times 4 \times 8 + 1 \times 8 = 1366.$$

Odd question: for what value of `i` does `x[i]` have the same address as `a[2][1]`?

Answer. That is,

$$1234 + 4 \times i = 1366$$
$$4 \times i = 132$$
$$i = 33$$

Of course, i is out of range.

14.5 No bulk assignments, and pointers previewed

The *value* of an array `a` is an address. After the 3rd quiz we shall study other kinds of variable whose value is an address: they are *pointers*.

In fact,

```
int main ( int argc, char * argv[] )
```

declares `argv` to be an array of pointers. More about this later.

But you cannot assign the value of one array to another.

```
int a[10], b[10];
a = b;
```

is an error, because the value of `b` is a *constant* address, as is the value of `b`.

```
a = b;
```

makes no sense; it would be like

```
3 = 4;
```