SCC.121: Fundamentals of Computer Science Two-Dimensional Arrays

Amit Chopra

amit.chopra@lancaster.ac.uk

Scenario

- We have built a **language ID** tool – a classification model for predicting the language of a given string.
- We want to track the performance of the classifier, particularly which languages are predicted best, and which languages are confused for others.
- Say we have strings in some languages. For strings in a given language, we need to count how many predictions are for each of those languages.

Possible approach

- Our tool will predict between 24 languages
  - Bulgarian, Croatian, Czech, Danish, Dutch, English, Estonian, Finnish, French, German, Greek, Hungarian, Irish, Italian, Latvian, Lithuanian, Maltese, Polish, Portuguese, Romanian, Slovak, Slovenian, Spanish and Swedish.

- We could make a list of triples containing: the actual language, the predicted language, and the count of predictions, and store these in a table.
  - {Portuguese, Portuguese, 569}
  - {Portuguese, Spanish, 32}
  - {Spanish, Portuguese, 24}
  - {Spanish, Spanish, 743}

Actual	Predicted	Count
Portuguese	Portuguese	569
Portuguese	Spanish	32
Spanish	Portuguese	24
Spanish	Spanish	743
…	…	…

Problems with approach

- A big table (24 languages * 24 rows, 576 rows), with repeated labels.
- Cumbersome to lookup information
  - Suppose I wanted to count the number of Actual Portuguese strings. Then I would look up each row in which ‘Portuguese’ appears in the Actual column and sum up the Count values for those rows.

Actual	Predicted	Count
Portuguese	Portuguese	569
Portuguese	Spanish	32
Spanish	Portuguese	24
Spanish	Spanish	743
…	…	…

Better Solution: A Matrix

	bg	hr	cs	da	nl	en	…	pt	ro	sk	sl	es	sv
bg													
hr													
cs													
da													
nl													
en													
…													
pt													
ro													
sk													
sl													
es													
sv													

- To find the counts for languages predicted for a given language, you look for the appropriate row. E.g. if I wanted to find da predictions for bg, pick row 0 and then pick column 3
- To find the actual language counts of predictions, you look for the appropriate column.
- Total number (of strings) in a language is the sum of the row.
- Total number of predictions of a language is the sum of the column.

Language	Code	No.
Bulgarian	bg	0
Croatian	hr	1
Czech	cs	2
Danish	da	3
Dutch	nl	4
English	en	5
Estonian	et	6
Finnish	fi	7
French	fr	8
German	de	9
Greek	el	10
Hungarian	hu	11
Irish	ga	12
Italian	it	13
Latvian	lv	14
Lithuanian	lt	15
Maltese	mt	16
Polish	pl	17
Portuguese	pt	18
Romanian	ro	19
Slovak	sk	20
Slovene	sl	21
Spanish	es	22
Swedish	sv	23

Representing our solution in code

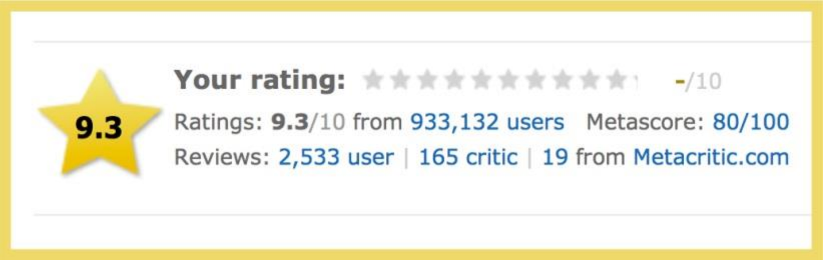
- All the counts are integers.

- We can label each language with an integer

- What we require now is a two-dimensional array where each element is an integer.

2-D Arrays

- Elements of Two-dimensional (2D) arrays are accessed by two indexes, one for the row and one for the column.

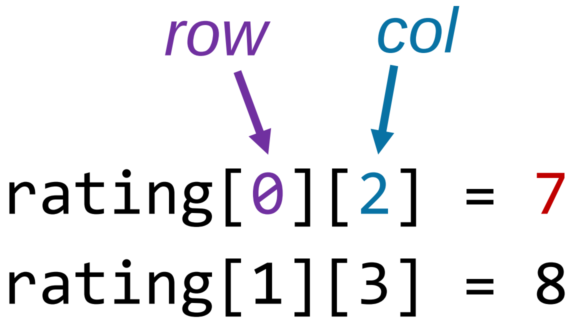


9.3

Your rating: ★★★★★★★★★★ -/10

Ratings: 9.3/10 from 933,132 users Metascore: 80/100

Reviews: 2,533 user | 165 critic | 19 from Metacritic.com



```
rating[0][2] = 7
rating[1][3] = 8
```

rating

movie (2nd index)

reviewer (1st index)

	0	1	2	3
0	4	6	7	5
1	7	9	4	8
2	6	9	3	7

Creating 2-D Arrays

- To create this empty array in C:

```
int rating[3][4];
```

	0	1	2	3
0				
1				
2				

- To create this empty array in Java:

```
int[][] rating = new int[3][4];
```

Another Example

```
char c[2][3];
```

c	0	1	2
0			
1			

- **`c`** has 2 rows, indexed 0 to 1, and 3 columns, indexed 0 to 2.

- Each element is of type **`char`**.

- Just as with linear arrays, we need to be able to access each individual element.

Array Indices

```
char c[2][3];
```

- Indices of 2-D array **c**.

c	0	1	2
0	[0][0]	[0][1]	[0][2]
1	[1][0]	[1][1]	[1][2]

- How can we generate each pair of indices
  - Use a “double loop”, or a loop within a loop.

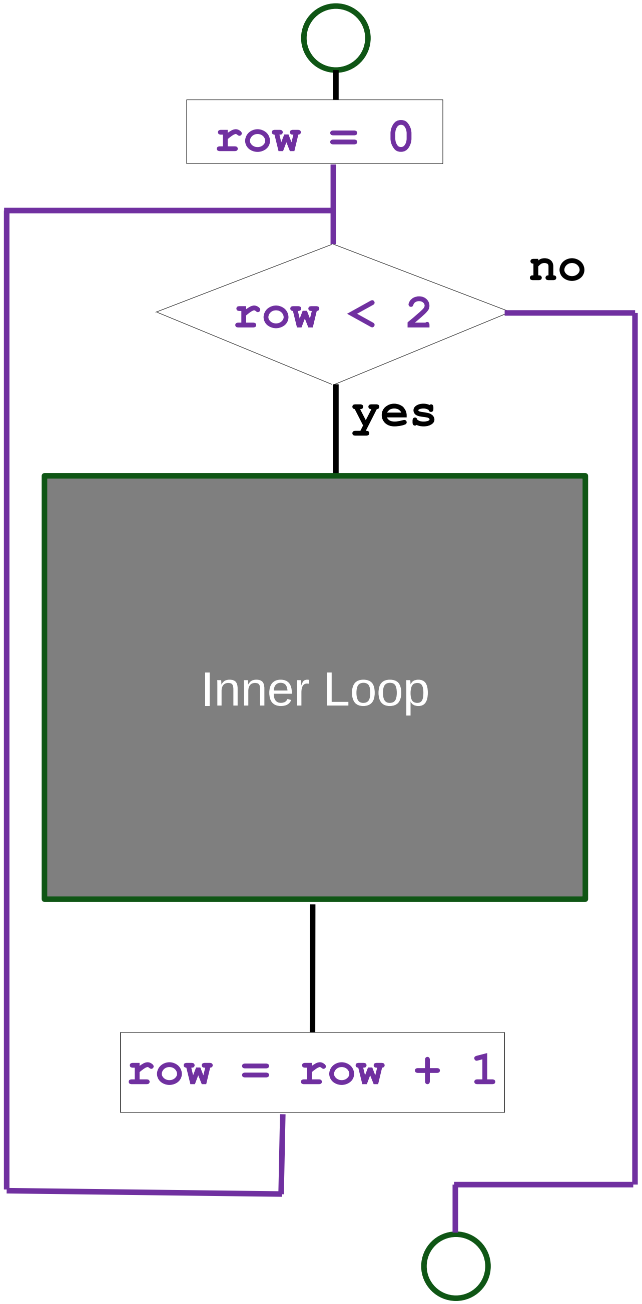
Double loop

```
char c[2][3];



for (int row = 0; row < 2; row++) {
    for (int col = 0; col < 3; col++) {
        print row, col
    }
}
```

Outer Loop

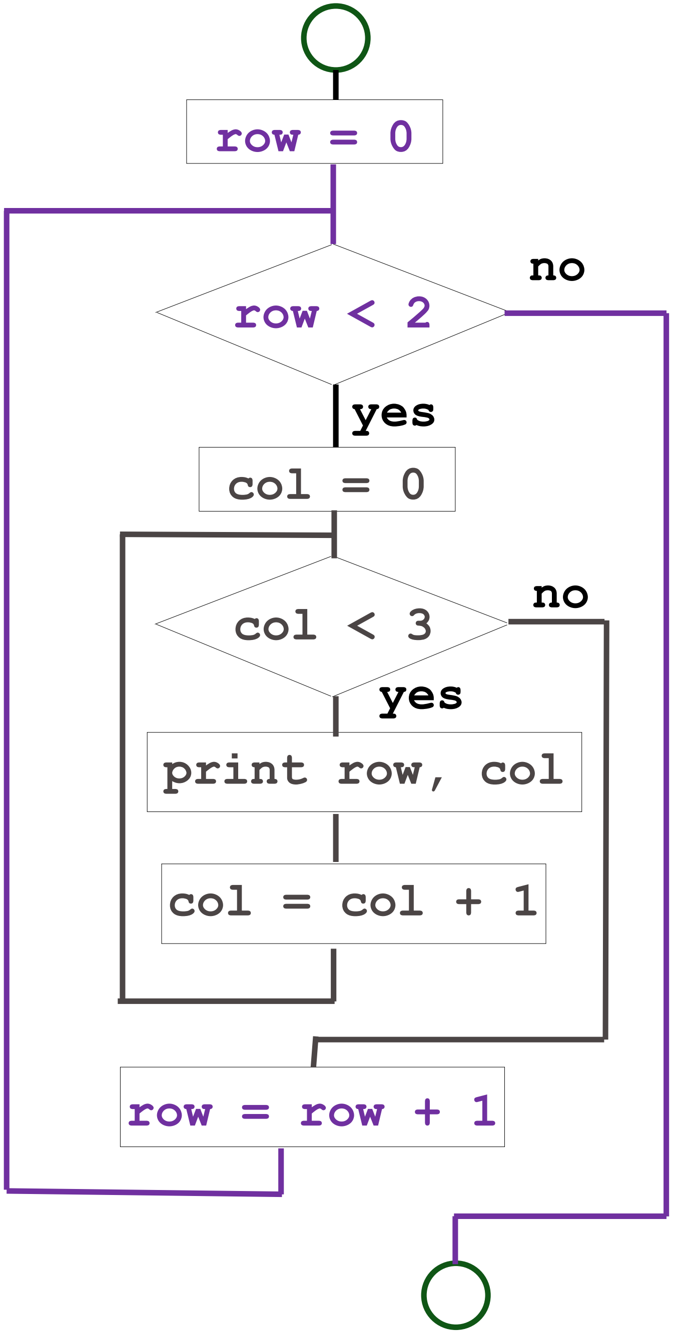
```
for (int row = 0; row < 2; row++) {
    for (int col = 0; col < 3; col++) {
        print row, col
    }
}
```

```
char c[2][3];
```



Outer and Inner Loops

```
for (int row = 0; row < 2; row++) {
    for (int col = 0; col < 3; col++) {
        print row, col
    }
}


char c[2][3];
```



Revisit: Our Language ID stats problem

	bg	hr	cs	da	nl	en	…	pt	ro	sk	sl	es	sv
bg													
hr													
cs													
da													
nl													
en													
…													
pt								569				32	
ro													
sk													
sl													
es								24				743	
sv													

2-D Array for matrix

- Here we set up the 2-D array to be used as our data structure.

- This will be a 24*24 array.

We’ll come back to this later

```
#define NUMBER_OF_LANGS 24

int counts[NUMBER_OF_LANGS][NUMBER_OF_LANGS];
```

Initialising the 2-D array

- This code uses a “double loop” to ensure that every element of the `stats` array is set to zero. And we add some dummy data.

```
void initCounts()
{
  for (int r = 0; r < NUMBER_OF_LANGS; r++) {
    for (int c = 0; c < NUMBER_OF_LANGS; c++) {
       counts[r][c] = 0;
    }
  }

  counts[18][18] = 569;
  counts[18][22] = 32;
  counts[22][18] = 24;
  counts[22][22] = 743;
}
```

Examining elements of 2-D array

- This code also uses a double loop to examine each element of the `stats` array.

- If the value of the element is not zero, it prints out the values of the two indices (r and c) and the value of the element.

```
void printCounts()
{
  for (int r = 0; r < NUMBER_OF_LANGS; r++) {
    for (int c = 0; c < NUMBER_OF_LANGS; c++) {
        if (counts[r][c] != 0)
          printf("%d, %d = %d\n", r, c, counts[r][c]);
    }
  }
}
```

Summing the total

- This code sums the total number of samples in our counts 2-D array.

```
int getTotal()
{
    int total = 0;
    for (int r = 0; r < NUMBER_OF_LANGS; r++) {
        for (int c = 0; c < NUMBER_OF_LANGS; c++) {
            total = total + counts[r][c];
        }
    }
    return total;
}
```

Summing the correct values

- Here we only look where the prediction is correct, i.e. where the row and column are equal.

```
int getTotalCorrect()
{
        int correct = 0;
        for (int l = 0; l < NUMBER_OF_LANGS; l++) {
                correct = correct + counts[l][l];
        }
        return correct;
}
```

Accuracy

- To find the accuracy of our classifier we just divide the total correct by the total number of samples.

```
void printAccuracy()
{
        int total = getTotal();
        int totalCorrect = getTotalCorrect();

        printf("\nAccuracy: %d/%d: %.2f%%\n", totalCorrect,
total, ((double) totalCorrect / (double) total) * 100.0);
}
```

```
Accuracy: 1312/1368: 95.91%
```

Language report

- We can look at a specific language by looking at its row and column separately.

```
void printReport(int lang)
{
        int samples = 0;
        for (int c = 0; c < NUMBER_OF_LANGS; c++) {
                samples = samples + counts[lang][c];
        }
        int predicted = 0;
        for (int r = 0; r < NUMBER_OF_LANGS; r++) {
                predicted = predicted + counts[r][lang];
        }

        int correct = counts[lang][lang];

        printf("Precision: %d/%d: %.2f%%\n", correct, predicted, ((double)
correct/ (double) predicted)*100.0);
        printf("Recall: %d/%d: %.2f%%\n", correct, samples, ((double)
correct/ (double) samples)*100.0);
}
```

Testing our code

```
printf("Counts package
testbed running\n");
initCounts();
printCounts();
printAccuracy();
printf("Portuguese:\n");
printReport(18);
printf("Spanish:\n");
printReport(22);
printf("application
terminated\n");
```

```
Counts package testbed running
18, 18 = 569
18, 22 = 32
22, 18 = 24
22, 22 = 743
Accuracy: 1312/1368: 95.91%
Portuguese:
Precision: 569/593: 95.95%
Recall: 569/601: 94.68%
Spanish:
Precision: 743/775: 95.87%
Recall: 743/767: 96.87%
application terminated
```

Encoding

- To find out what the languages are, we can use the encoding from earlier, and decode our stats about languages 18 and 22.

- Not very nice for us to have to do this

Language	Code	No.
Bulgarian	bg	0
Croatian	hr	1
Czech	cs	2
Danish	da	3
Dutch	nl	4
English	en	5
Estonian	et	6
Finnish	fi	7
French	fr	8
German	de	9
Greek	el	10
Hungarian	hu	11
Irish	ga	12
Italian	it	13
Latvian	lv	14
Lithuanian	lt	15
Maltese	mt	16
Polish	pl	17
Portuguese	pt	18
Romanian	ro	19
Slovak	sk	20
Slovene	sl	21
Spanish	es	22
Swedish	sv	23

Arrays: Pros

- We can use a single name to represent multiple data items of the same type.
- **Random access** – very fast
  - We can pick a valid index (at “random”) from the index range of an array and very quickly “access” the value stored at that position within the array.
  - As opposed to sequentially going through the array.

Arrays: Cons

- **Arrays are of fixed size – cannot be resized**
  - If it is possible to dynamically allocate space, we can allocate bigger arrays but there is an associated cost of copying elements overs.

- **Insertions and deletions from arrays are costly**
  - A similar cost to the one indicated above; we will see an example of this in our study of Vectors.

#define – “manifest constants”

- What’s this about?

```
        #define NUMBER_OF_LANGS 24
```

- This allows us to associate a (meaningful) name or label with a value.

- This is NOT a variable! It is a **constant**.

- Why is this useful?

Usefulness of constants

- Say the number of languages our program has to handle changes.

- Without the NUMBER_OF_LANGS constant, we would have to search our lines of code for the value 24 and change it to the new value.

Search & Replace

```
void initCounts()
{
       for (int r = 0; r < 24; r++)
         for (int c = 0; c < 24; c++){
                      counts[r][c] = 0;
       }
}
void printCounts()
{
   for (int r = 0; r < 24; r++)
      for (int c = 0; c < 24; c++)
      {
          if (counts[r][c] != 0)
              printf("%d, %d = %d\n", r, c, counts[r][c]);
      }
}
```

Danger

- Always the danger that we may have the value 24 appearing somewhere in the code where it doesn’t represent the number of teams.
- So we might change it and perhaps the code no longer works as expected.
- Or we might accidentally decide not to change a 24 where we should have…
- Classic example: a change in VAT rate.

“manifest constants”

- If we use constants correctly, if the number of languages changes then we only have to change one line of code …

```
#define NUMBER_OF_LANGS 30
```

- … recompile our code and the program should continue to function as expected.
- Even if we never have to change the number of languages, using a label or meaningful name increases the readability (and therefore understandability) of our code.