Arrays - P2

CS1A

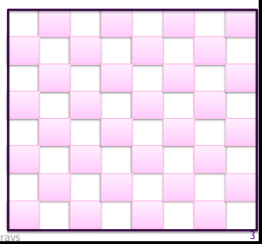
- Multi-Dimensional Arrays

Multi-Dimensional Arrays

- So far, we've discussed one-dimensional arrays
- It is possible to have arrays of more than 1 dimension → multi-dimensional arrays
  - There is one subscript for each dimension
    - A 2-dimensional array has 2 subscripts
    - A 3-dimensional array has 3 subscripts … and so on
  - Think of them as an array of arrays
  - For example, an array of c-strings
    - c-strings are an array

Example - Chessboard

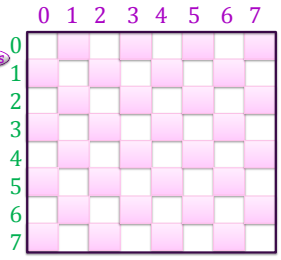
- Creating a chessboard program
  - You want to track the pieces
- We could do a one dimensional array
  int board[64];
- A two-dimensional array would make more sense for this application
  int board[8][8];
- A 2-dimensional array for this application better corresponds to the real-world application



- This way we could think of the array in terms of (x, y) pairs.
  x could represent the rows and
  y could represent the columns

int board[8][8];

Of Rows # Of Columns



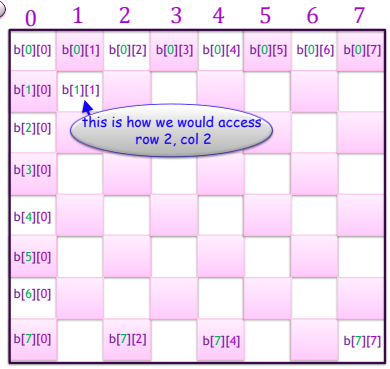
Chessboard Example

int b [8][8];

Rows Cols

This way we can reference our array with respect to the rows and columns

So we can say b[1][1] instead of b[9]



Example 2 – Scores

- Let's say you are tracking a group of scores from different people
- Let's say we are tracking 2 people with 3 items to score
  We declare our array like this → int scoresAr[2][3];

1st Person
scoresAr[0][0] = 75
scoresAr[0][1] = 65
scoresAr[0][2] = 95

2nd Person
scoresAr[1][0] = 45
scoresAr[1][1] = 85
scoresAr[1][2] = 100

	1st Score	2nd Score	3rd Score
1st person	75	65	95
2nd person	45	85	100

Initializing Multidimensional Arrays

- We initialize multidimensional arrays a little differently

```
int scoresAr[2][3] = { 75, 65, 95, 45, 85, 100 };
⇔int scoresAr[2][3] = {  { 75, 65,  95 },
                         { 45, 85, 100 }  };
```

- Although these are equivalent the 2nd is easier to read
  - The compiler ignores the extra brackets, but needs the commas
- Or we can initialize all values to 0 like this:

```
int scoresAr[2][3] = {0};
```

Again.. We should use constants where we can:

```
const int TOTAL_PLAYERS = 2;
const int TOTAL_SCORES  = 3;

int scoresAr[TOTAL_PLAYERS][TOTAL_SCORES] = {0};
```

Generally speaking we should always initialize arrays

Using For loops

```
int scoresAr[2][3];

for (int i = 0; i < 2; i++)
{
  cout << "Enter scores for player #" << i + 1 << " : ";
  for (int j = 0; j < 3; j++)
  {
     cout << "Enter score #" << j + 1 << " : ";
     cin  >> scoresAr [i][j];
  }
}
```


Looping through rows

scoresAr

player	score [0]	score [1]	score [2]
[0]	50	70	90
[1]	60	80	100

player score totalScore avgScore

```
const int TOTAL_PLAYERS = 2;
const int TOTAL_SCORES  = 3;

int scoresAr[TOTAL_PLAYERS][TOTAL_SCORES] = { {50, 70,  90},
                                              {60, 80, 100} };
int totalScore, player, score;
float avgScore;

for (player = 0; player < TOTAL_PLAYERS;  player++)
{
   totalScore = 0;
   for (score = 0; score < TOTAL_SCORES; score++)
   {
     totalScore += scoresAr[player][score];
   }
   avgScore = float(totalScore) / TOTAL_SCORES;
   cout << "Average for player #" << player + 1 << " = " << avgScore << endl;
```

OUTPUT

What does this do? Do a desk check

Looping through columns

Write the code to output the average score per round -Desk check your code

Passing 2-D arrays as Parameters

```
int scoresAr[TOTAL_PLAYERS][TOTAL_SCORES] = {0};
float avg;
int player, score;

AverageArray(scoresAr);

float AverageArray(int intAr[][TOTAL_SCORES], int player)
{
    int sum;
    sum = 0;
    for (player = 0; player < TOTAL_PLAYERS; player++)
    {
       for (score = 0; score < TOTAL_SCORES;  score++)
       {
          sum += arrayValues[player][score];
       }
       cout << (float(sum)/ TOTAL_SCORES);
    }
    cout << "Average for player #" << player + 1 << " = ";
    cout << AverageArray(scoresAr, player);
}
```

How should TOTAL_SCORES be declared?

It should be passed by Const reference

You do not need to specify the 1st dimension. You do need to specify the 2nd dimension

C++ doesn't need to know how many rows, just the number of elements in each row.

2D - Arrays

- How can C++ figure out the address in a 2-D Array?
  - Let's say we wanted to create a word scramble game. We would use a 2-D array to store the letters: `char boardAr[4][4];`

Base Address 115 — 1 byte

	[0]	[1]	[2]	[3]
[0]	115 N	116 A	117 P	118 P
[1]	119 A	120 O	121 L	122 I
[2]	123 N	124 E	125 M	126 U
[3]	127 G	128 A	129 B	130 P

This is why we have to pass the column - C++ uses it to calculate the address of the element we need. Why don't we have to pass The row size?

Row index * # of Col takes it to this row
115 + (2 * 4) = 123

- How can it calculate boardAr[2][1]'s address?

Base Address | Row Index number | # of Columns | Col Index number | # of Bytes In datatype

115 +(2 * 4 + 1) * 1 = 124

2D - Arrays

- What if we accidentally did this: cout << board[0][5];?
  - char boardAr[4][4];
    1 byte

- How would it calculate boardAr[0][5]'s address?

Base Address | Row Index number | # of Columns | Col Index number | # of Bytes In datatype

+(* +) * =

How does C++ see the array?

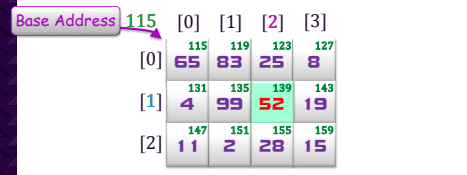
- It simply sees the base address and calculates then element based
  - the row Index
  - the column index
  - and the # of columns

- The column size tells C++ where the next row is...
- The row size doesn't matter..
  - Because it doesn't calculate the address of each element based on the row size.
  - char boardAr[4][4];

Any 2D array can be represented as a 1D array - we use 2D arrays Because it makes it easier for the programmer to understand.

2D - Arrays

- Let's try that with a different data type
  - What if instead it was a 2D array of integers? int boardAr[3][4];
    4 bytes



- How can it calculate boardAr[1][2]'s address?

Base Address | Row Index number | # of Columns | Col Index number | # of Bytes In datatype

115 +(1 * 4 + 2) * 4 = 139

Passing one row from a 2D Array

The previous example illustrates passing a 2D array into a 2D array

- but does average player need the entire array?
  NO - it only needs the data for 1 player or 1 row
- If we only need to access 1 row - we can pass in that one row into a 1D array

```
...
for (player = 0; player < TOTAL_PLAYERS; player++)
{
    cout << "Average for player #" << player + 1 << " = ";
    cout << AverageIntAr(scoresAr[player ], TOTAL_SCORES);
}

float AverageIntAr(const int INT_AR[], const int AR_SIZE)
{
    for (int index = 0; index < AR_SIZE;  index++)
    {
        sum += INT_AR[index];
    }
    return (float(sum)/ AR_SIZE);
```

We don't need to pass this it is Global - but it allows us to use th Generic function

This passes in the row's address - into the 1D array

In here, we only have access to 1 players scores

Enums & Arrays

- Enums and Arrays work really well together when trying to output an enum as a string type.
  - Map the Enum values to the array indices.

For example:

```
enum Color
{
  RED,
  GREEN,
  BLUE
};

const int MAX_COLORS = 3;
const string COLOR_AR[MAX_COLORS] = {"red", "green","blue"};
…
<in a function or main()>
Color myColor;
myColor = RED;

cout << COLOR_AR[myColor];
```

This will output "red"

Exercise

	Color #1	Color #2	Color #3
COOL	blue	teal	indigo
WARM	purple	green	brown
HOT	red	yellow	orange

Write a program that will help users determine which colors in the color palette based on whether they want cool, warm or hot colors.

```
blue
teal
indigo

purple
green
brown

red
yellow
orange
```

First let's use enumerated types for the color spectrums.

```
enum Spectrum
{
    COOL,
    WARM,
    HOT
};
```

INPUT FILE

1. Read the data into a 2D Array from a file. *(make this a generic function)*
2. Read and validate the input - use a menu *(make this a generic function)*
3. Convert ***Int*** input into ***Spectrum*** type
4. Output one row of the array based on the selection
5. Output the array in table format (see above)

MENU

```
**********************
  SPECTRUM SELECTOR
**********************
(0) Cool
(1) Warm
(2) Hot
(3) Output All
(4) Exit
Select a spectrum:
```