# Computer Science 1: Starting Computing
# CSCI 1300

University of Colorado Boulder

# Computer Science 1: Starting Computing CSCI 1300

Dr. Ioana Fleming
Spring 2019
Lecture 18

University of Colorado Boulder

# Reminders

Submissions:

- Homework 6: due Monday 3/4 at 6pm
- Homework 7: due Wednesday 3/13 at 6pm

Readings:

- Ch. 6 – Arrays – 2D
- Ch. 8 – Streams

To discuss:

- Global Variables

© traveler1116/iStockphoto.

# Chapter Six: Arrays and Vectors

# Topic 2

1. Arrays
2. Common array algorithms
3. Arrays / functions
4. Problem solving: adapting algorithms
5. Problem solving: discovering algorithms
6. 2D arrays
7. Vectors
8. Chapter Summary

# Common Algorithms – Copying

Consider these two arrays:

```
int squares[5] = { 0, 1, 4, 9, 16 };
int lucky_numbers[5];
```

How can we copy the values from **`squares`** to **`lucky_numbers`**?

Let's try what seems right and easy…

```
squares = lucky_numbers;
```

…and wrong!

*You cannot assign arrays!*

*The compiler will report a syntax error.*

# Common Algorithms – Copying Requires a Loop

```
/* you must copy each element individually using a
loop! */

int squares[5] = { 0, 1, 4, 9, 16 };
int lucky_numbers[5];

for (int i = 0; i < 5; i++)
{
   lucky_numbers[i] = squares[i];
}
```

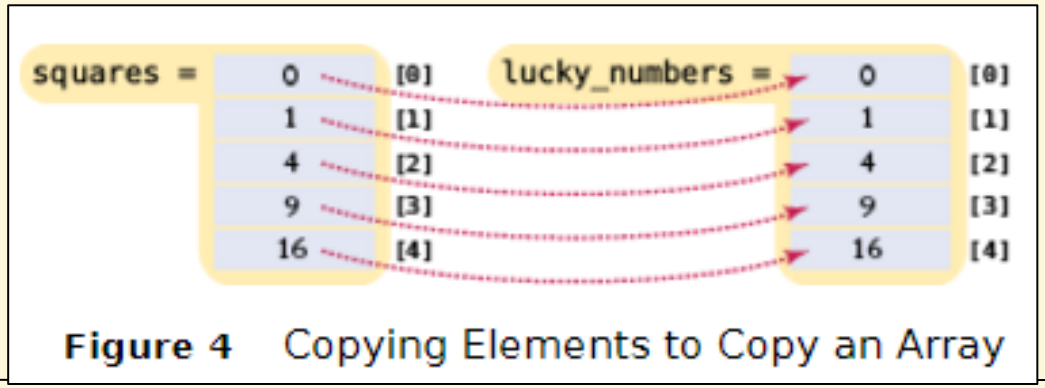



**Figure 4** Copying Elements to Copy an Array

# Common Algorithms – Sum and Average Value

You have already seen the algorithm
for computing the sum and average of a set of data.
The algorithm is the same when the data is stored in an array.

```
double total = 0;
for (int i = 0; i < size; i++)
{
   total = total + values[i];
}
```

The average is just arithmetic:

```
double average = total / size;
```

# Common Algorithms – Minimum

For the minimum, we just reverse the comparison.

```
double smallest = values[0];
for (int i = 1; i < size; i++)
{
   if (values[i] < smallest)
   {
      smallest = values[i];
   }
}
```

These algorithms require that the array contain at least one element.

# Common Algorithms – Linear Search

Find the position of a certain value, say 100, in an array:

```
int pos = 0;
bool found = false;
while (pos < size && !found)
{
   if (values[pos] == 100) // looking for 100
   {
      found = true;
   }
   else
   {
      pos++;
   }
}
```

# Common Algorithms – Removing an Element, Unordered

To remove the element at index **i**:

If the elements in the array are not in any particular order, simply overwrite the element to be removed with the *last* element, then shrink the size by 1.

```
values[pos] = values[current_size - 1];
current_size--;
```

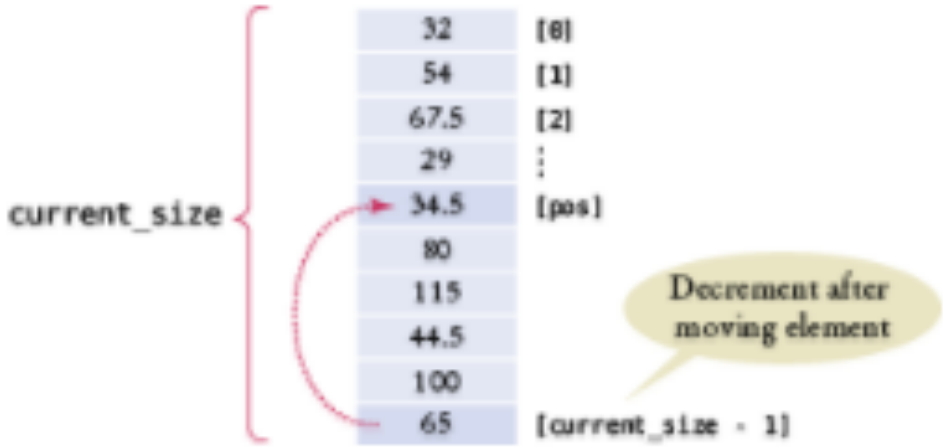



**Figure 5** Removing an Element in an Unordered Array

# Common Algorithms – Removing an Element, Ordered

The situation is more complex if the order of the elements matters.

Then you must move all elements following the element to be removed "down" (to a lower index), and then shrink the size of the vector by removing the last element.

```
for (int i = pos + 1; i < current_size; i++)
{
   values[i - 1] = values[i];
}
current_size--;
```

# Common Algorithms – Removing an Element, Ordered

```
//removing the element at index "pos"
for (int i = pos + 1; i < current_size; i++)
{
   values[i - 1] = values[i];
}
current_size--;
```

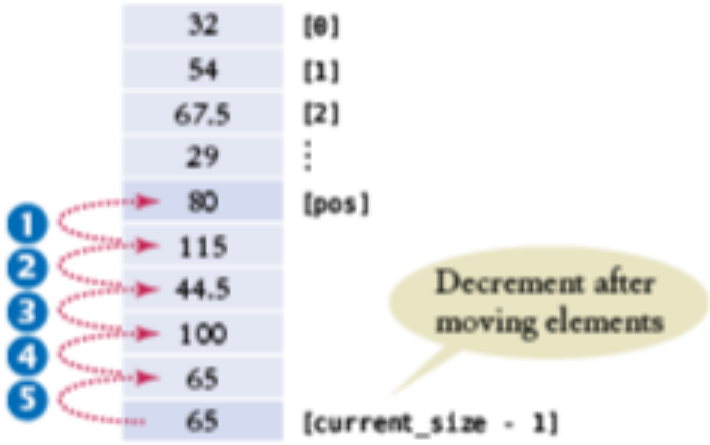



**Figure 6** Removing an Element in an Ordered Array

# Common Algorithms – Inserting an Element Unordered

If the order of the elements does not matter, in a partially filled array (which is the only kind you can insert into), you can simply insert a new element at the end.

```
if (current_size < CAPACITY)
{
   current_size++;
   values[current_size - 1] = new_element;
}
```

# Common Algorithms – Inserting an Element Ordered

If the order of the elements *does* matter, it is a bit harder.

To insert an element at position **i**, all elements from that location to the end of the vector must be moved "out" to higher indices.

After that, insert the new element at the now vacant position **[i]**.

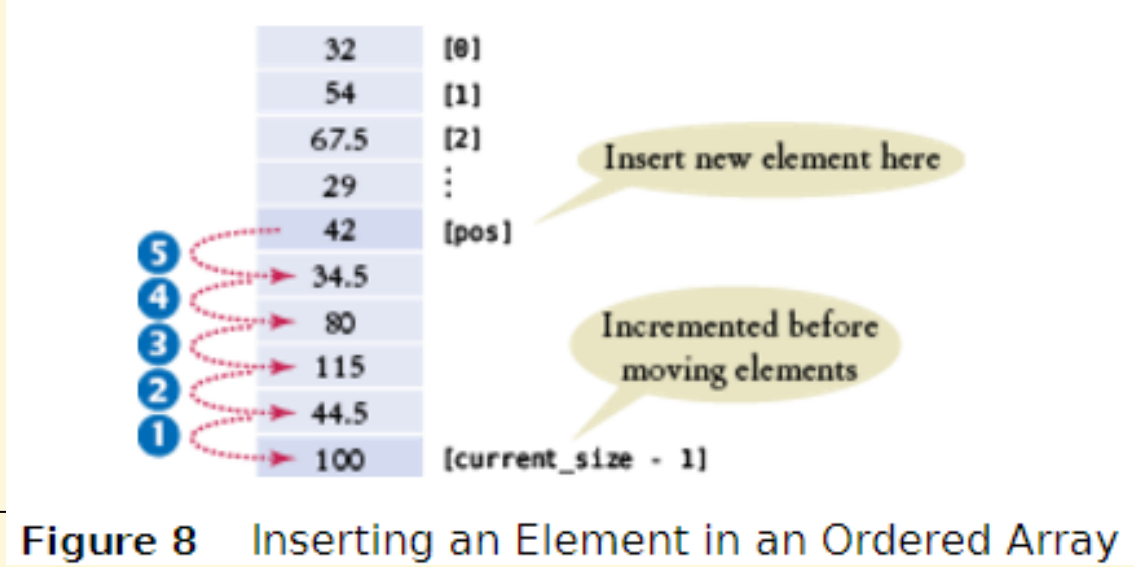



**Figure 8** Inserting an Element in an Ordered Array

# Inserting an Element Ordered: Code

```
if (current_size < CAPACITY)
{
   current_size++;
   for (int i = current_size - 1; i > pos; i--)
   {
      values[i] = values[i - 1];
   }
   values[pos] = new_element;
}
```

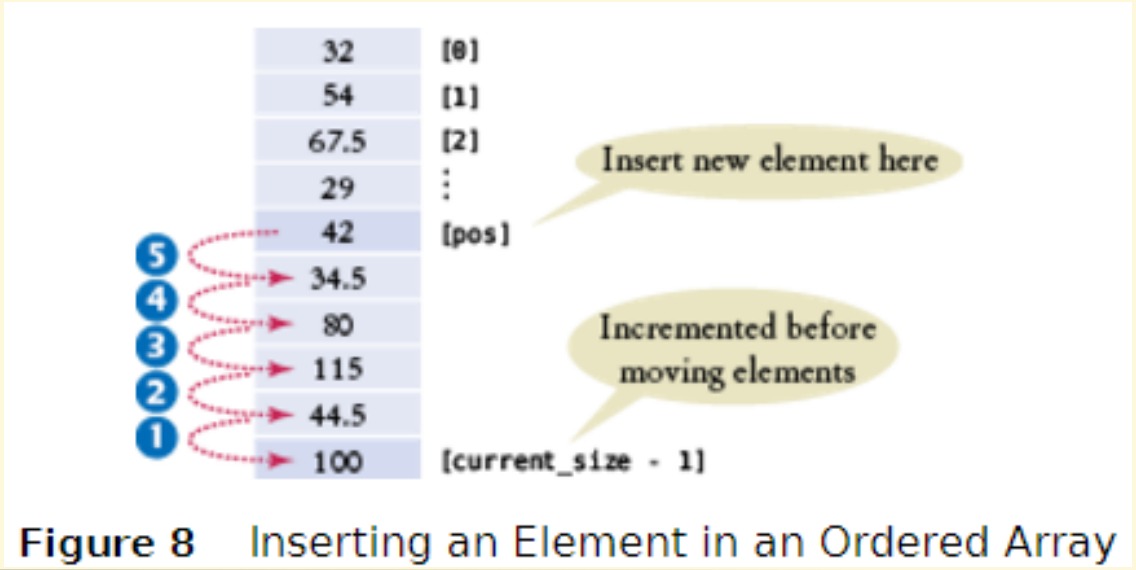



**Figure 8** Inserting an Element in an Ordered Array

# Common Algorithms – Swapping Elements

Suppose we need to swap the values at positions **i** and **j** in the array.
Will this work?

```
values[i] = values[j];
values[j] = values[i];
```

Look closely!

In the first line you lost – forever! – the value at **i**, replacing it with the value at **j**.

Then what?
Put' **j**'s value back in **j** in the second line?

We end up with 2 copies of the [j] value, and have lost the [i]

# Code for Swapping Array Elements

```
//save the first element in
//  a temporary variable
// before overwriting the 1st

double temp = values[i];
values[i] = values[j];
values[j] = temp;
```

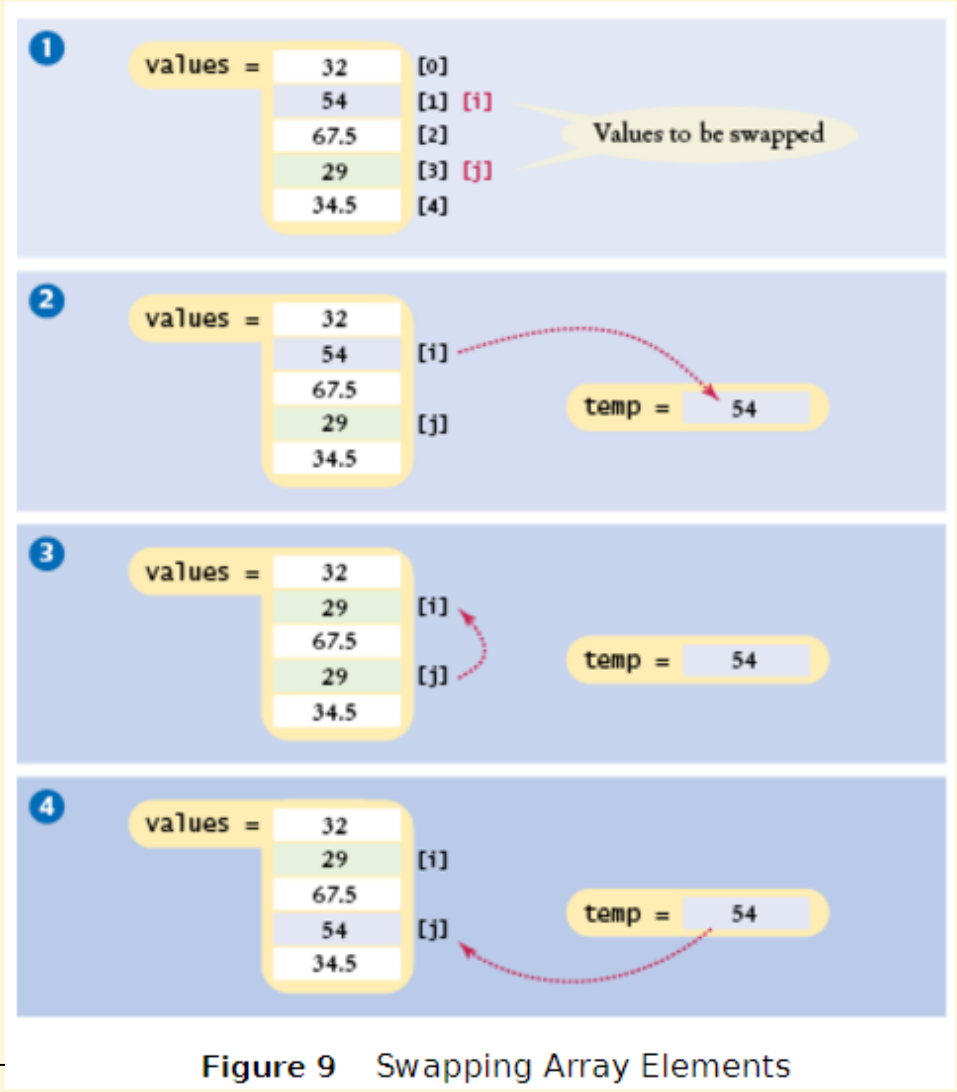



**Figure 9** Swapping Array Elements

# Topic 3

1. Arrays
2. Common array algorithms
3. Arrays / functions
4. Problem solving: adapting algorithms
5. Problem solving: discovering algorithms
6. 2D arrays
7. Vectors
8. Chapter Summary

# Arrays as Parameters in Functions

Recall that when we work with arrays we use a companion variable.

The same concept applies when using arrays as parameters:

You must pass the size to the function so it will know how many elements to work with.

# Array as function argument

- What does the computer know about an array?
  - The base type
  - The address of the first indexed variable
  - The number of indexed variables
- What does a function know about an array argument?
  - The base type
  - The address of the first indexed variable

# Entire Arrays as Arguments

- Formal parameter can be entire array
  - Argument then passed in function call is array name
  - Called "array parameter"
- Send size of array as well
  - Typically done as second parameter
  - Simple int type formal parameter

# Array Parameter Function Example

Here is the **sum** function with an array parameter:
Notice that to pass one array, it takes two parameters.

```
double sum(double data[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++)
   {
      total = total + data[i];
   }
   return total;
}
```

# Entire Array as Argument Example

- In some main() function definition, consider this calls:

```
int score[5], numberOfScores = 5;
fillup(score, numberOfScores);
```

  - 1st argument is entire array
  - 2nd argument is integer value
- Note no brackets in array argument!

# Array Parameters

- May seem strange
  - No brackets in array argument
  - Must send size separately
- One nice property:
  - Can use SAME function to fill any size array!
  - Exemplifies "re-use" properties of functions
  - Example:

```
int score[5], time[10];
fillUp(score, 5);
fillUp(time, 10);
```

# Array as Argument: How?

- What's really passed?
- Think of array as 3 "pieces"
  - Address of first indexed variable (`arrName[0]`)
  - Array base type (`int` or `double` or `float` or …)
  - Size of array
- Only 1st piece is passed!
  - Just the beginning address of array (the 1st element)
  - Knowing the type helps us retrieve the (2nd – last) elements

# Array Parameters in Functions Require `[]` in the Header

You use an empty pair of square brackets *after* the parameter variable's name to indicate you are passing an array.

```
double sum(double data[], int size)
```

# Array Function Call Does NOT Use the Brackets!

When you call the function, supply both the name of the array and the size, BUT NO SQUARE BRACKETS!!

```
double NUMBER_OF_SCORES = 10;
double scores[NUMBER_OF_SCORES]
   = { 32, 54, 67.5, 29, 34.5, 80, 115, 44.5, 100, 65 };
double total_score = sum(scores, NUMBER_OF_SCORES);
```

You can also pass a smaller size to the function:

```
double partial_score = sum(scores, 5);
```

This will sum over only the first five **`double`**s in the array.

# Array Parameters Always are Reference Parameters

When you pass an array into a function, the contents of the array can ***always*** be changed. An array name is actually a reference, that is, a memory address:

```
   //function to scale all elements in array by a factor
void multiply(double values[], int size, double factor)
   {
      for (int i = 0; i < size; i++)
      {
         values[i] = values[i] * factor;
      }
   }
```

# Arrays as Parameters but No Array Returns

You can pass an array into a function but

you cannot return an array.

However, the function can modify an input array, so the function definition must include the result array in the parentheses if one is desired.

# Arrays as Parameters and Return Value

If a function can change the size of an array,
it should let the caller know the new size by returning it:

```
int read_inputs(double inputs[], int capacity)
{
   //returns the # of elements read, as int
    int current_size = 0;
    double input;
    while (cin >> input)
    {
       if (current_size < capacity)
       {
          inputs[current_size] = input;
          current_size++;
       }
    }
    return current_size;
}
```

# Array Parameters in Functions: Calling the Function

Here is a call to the **`read_inputs`** function:

```
const int MAXIMUM_NUMBER = 1000;
double values[MAXIMUM_NUMBER];
int current_size =
      read_inputs(values, MAXIMUM_NUMBER);
```

After the call, the **`current_size`** variable specifies how many were added.

# Function to Fill or Append to an Array

Or it can let the caller know by passing and returning the current size:

```
int append_inputs(double inputs[], int capacity,
                  int current_size)
{
   double input;
   while (cin >> input)
   {
      if (current_size < capacity)
      {
         inputs[current_size] = input;
         current_size++;
      }
   }
   return current_size;
}
```

*Note this function has the added benefit of either filling an empty array or appending to a partially-filled array.*

# Array Functions Example

The following program uses the preceding functions to read values from standard input, double them, and print the result.

- The **`read_inputs`** function fills an array with the input values. It returns the number of elements that were read.
- The **`multiply`** function modifies the contents of the array that it receives, demonstrating that arrays can be changed inside the function to which they are passed.
- The **`print`** function does not modify the contents of the array that it receives.

# Array Functions Example Code

- `score_v2.cpp`

# Constant Array Parameters

- When a function doesn't modify an array parameter, it is considered good style to add the `const` reserved word, like this:

```
double sum(const double values[], int size)
```

- The `const` reserved word helps the reader of the code, making it clear that the function keeps the array elements unchanged.

- If the implementation of the function tries to modify the array, the compiler issues a warning.

# The const Parameter Modifier

- Recall: array parameter actually passes address of 1st element
- Function can then modify array!
  - Often desirable, sometimes not!
- Protect array contents from modification
  - Use "const" modifier before array parameter
    - *Called "constant array parameter"*
    - *Tells compiler to "not allow" modifications*

# Example – function definition

```
void addarray(int size,          // IN size of arrays
              const float A[],   // IN input array
              const float B[],   // IN input array
              float C[])         // OUT result array

// Takes two arrays of the same size as input parameters
// and outputs an array whose elements are the sum of the
// corresponding elements in the two input arrays.
{
    int i;
    for (i = 0; i < size; i++)
         C[i] = A[i] + B[i];

  } // End of function addarray
```

# Example – function call

The function `addarray` could be used as follows:

In main():

```
int one[50], two[50], three[50];
            //
            //
addarray(50, one, two, three);

// but also:
addarray(20, one, two, three);

// it will only do the addition on the first 20 elements
of each array
```

# Topic 6

1. Arrays
2. Common array algorithms
3. Arrays / functions
4. Problem solving: adapting algorithms
5. Problem solving: discovering algorithms
6. 2D arrays
7. Vectors
8. Chapter Summary

# Two-Dimensional Arrays

It often happens that you want to store collections of values that have a two-dimensional layout.

Such data sets commonly occur in financial and scientific applications.

An arrangement consisting of *tabular data (rows and columns* of values) is called:

a ***two-dimensional array***, or a ***matrix***

# Two-Dimensional Array Example

Consider the medal-count data from the 2014 Winter Olympic skating competitions:

| **Country** | Gold | Silver | Bronze |
|---|---|---|---|
| Canada | 0 | 3 | 0 |
| Italy | 0 | 0 | 1 |
| Germany | 0 | 0 | 1 |
| Japan | 1 | 0 | 0 |
| Kazakhstan | 0 | 0 | 1 |
| Russia | 3 | 1 | 1 |
| South Korea | 0 | 1 | 0 |
| United States | 1 | 0 | 1 |

# Defining Two-Dimensional Arrays

C++ uses an array with *two* subscripts to store a *2D* array.

```
const int COUNTRIES = 8;
const int MEDALS = 3;
int counts[COUNTRIES][MEDALS];
```

An array with 8 rows and 3 columns is suitable for storing our medal count data.

# Defining Two-Dimensional Arrays – Initializing

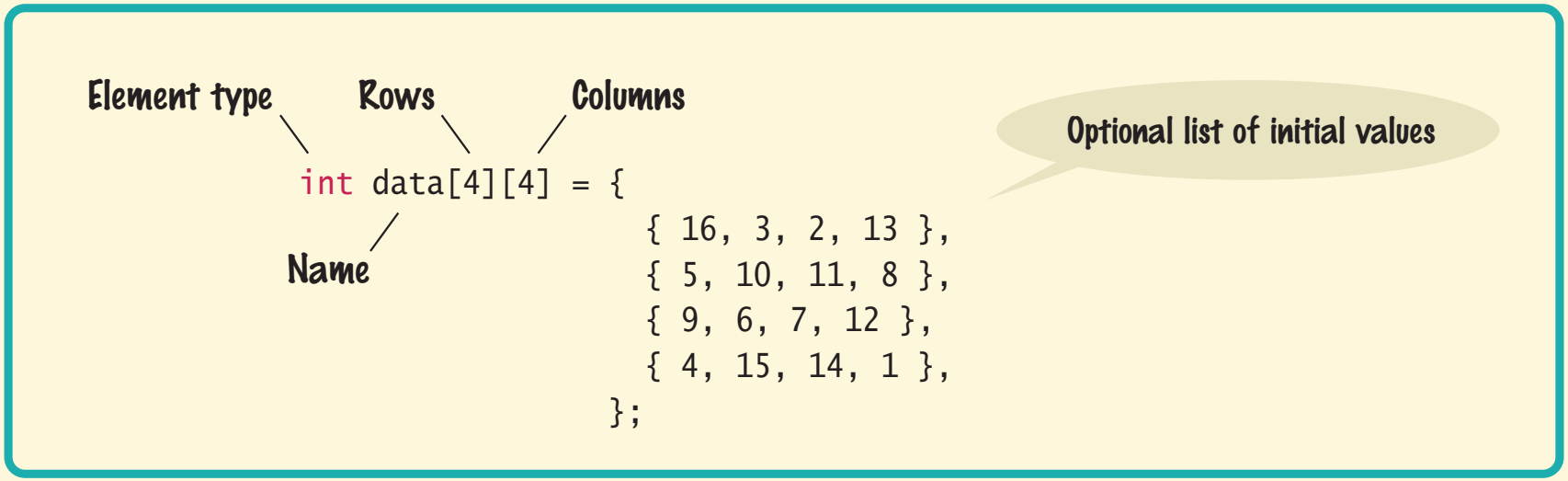
Just as with one-dimensional arrays, you *cannot* change the size of a two-dimensional array once it has been defined.

But you can initialize a 2-D array:

```
int counts[COUNTRIES][MEDALS] =
{
   { 0, 3, 0 },
   { 0, 0, 1 },
   { 0, 0, 1 },
   { 1, 0, 0 },
   { 0, 0, 1 },
   { 3, 1, 1 },
   { 0, 1, 0 }
   { 1, 0, 1 }
};
```

# Defining 2D arrays

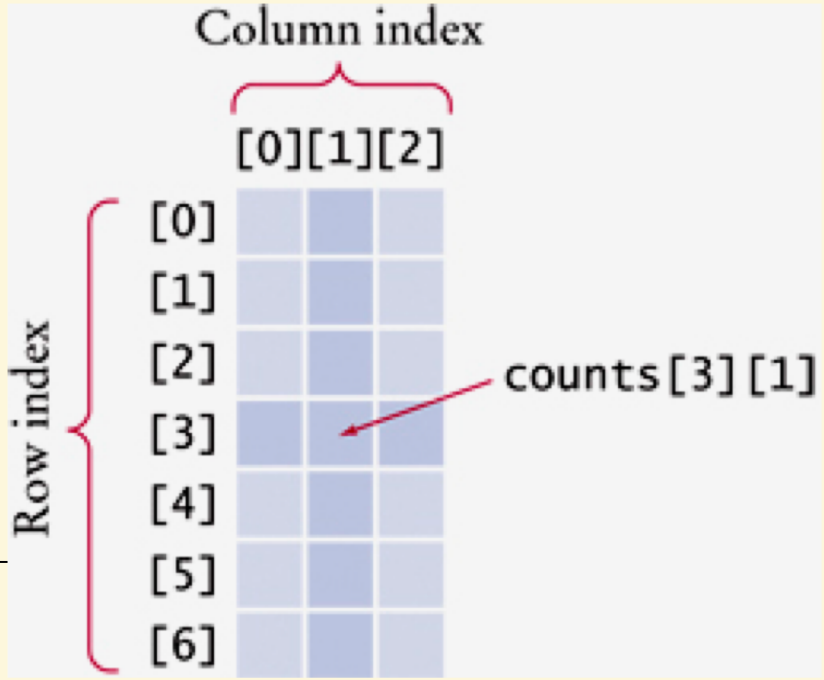
Two-Dimensional Array Definition



# 2D Arrays – Accessing Elements

```
// copy to value what is currently
// stored in the array at [3][1]
int value = counts[3][1];

// Then set that position in the array to 8
counts[3][1] = 8;
```



# Two-Dimensional Array - Printing

Consider the medal-count data from the 2014 Winter Olympic skating competitions:

| **Country** | Gold | Silver | Bronze |
|---|---|---|---|
| Canada | 0 | 3 | 0 |
| Italy | 0 | 0 | 1 |
| Germany | 0 | 0 | 1 |
| Japan | 1 | 0 | 0 |
| Kazakhstan | 0 | 0 | 1 |
| Russia | 3 | 1 | 1 |
| South Korea | 0 | 1 | 0 |
| United States | 1 | 0 | 1 |

# Print All Elements in a 2D Array

```
for (int i = 0; i < COUNTRIES; i++)
{
   // Process the ith row
   for (int j = 0; j < MEDALS; j++)
   {
      // Process the jth column in the ith row
      cout << setw(8) << counts[i][j];
   }
   // Start a new line at the end of the row
   cout << endl;
}
```

# Multidimensional Array Parameters

- Similar to one-dimensional array
  - 1st dimension size not given
    - Provided as second parameter
  - 2nd dimension size IS given
- Example:

```
void DisplayPage(const char p[][100], int sizeDimension1)
{
    for (int index1=0; index1<sizeDimension1; index1++)
    {
           for (int index2=0; index2 < 100; index2++)
                  cout << p[index1][index2];
           cout << endl;
    }
}
```

# 2D Array Parameter Columns Hardwired

That function works for only arrays of 3 columns.

If you need to process an array
with a different number of columns, like 4,

you would have to write
***a different function***
that has 4 as the parameter.

# Omitting the Column size of a 2D Array Parameter

When passing a one-dimensional array to a function, you specify the size of the array as a separate parameter variable:

```
void print(double values[], int size)
```

This function can print arrays of any size. However, for two-dimensional arrays you cannot simply pass the numbers of rows and columns as parameter variables:

```
void print(double table[][], int rows, int cols) //NO!
```

```
const int COLUMNS = 3;
void print(const double table[][COLUMNS], int rows) //OK
```

This function can print tables with any number of rows, but the column size is fixed.

# Two-Dimensional Array Storage

What's the reason behind this?

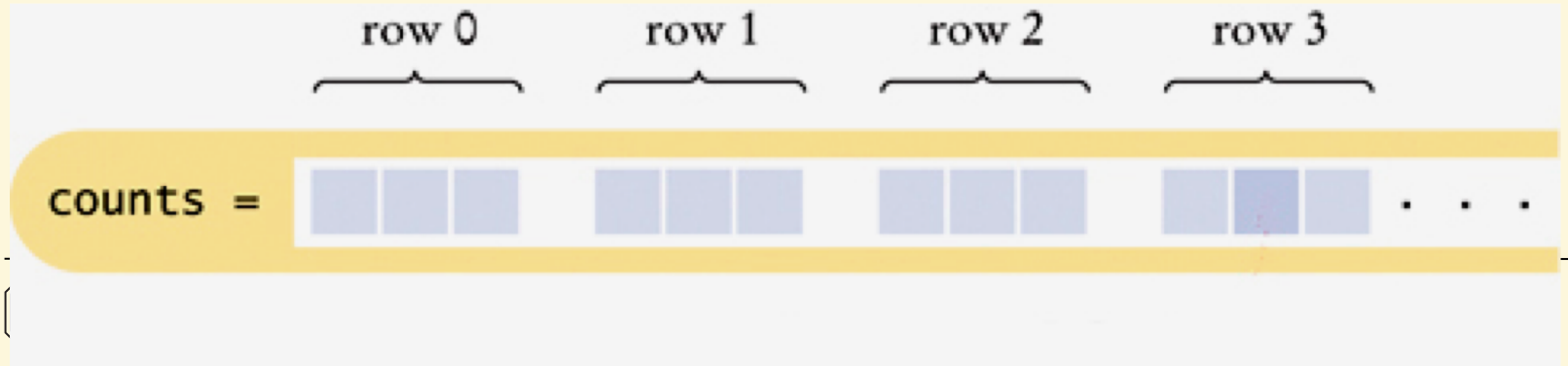
Although the array appears to be two-dimensional,
the elements are still stored as a linear sequence.

**`counts`** is stored as a sequence of rows, each 3 long.

So where is **`counts[3][1]`** ?

The offset (calculated by the compiler) from the start of the array is

**`3 x number of columns + 1`**



# 2D Array Parameters: Rows

The **`row_total`** function: does it need to know the number of rows of the array?

What about **`column_total()`?**

If the number of rows is required, pass it in:

```
int column_total(int table[][COLUMNS], int rows, int col)
{
   int total = 0;
   for (int i = 0; i < rows; i++)
   {
      total = total + table[i][col];
   }
   return total;
}
```

# Practice It: 2D Array Parameters

Insert the missing statement. The function should return the result of adding the values in the first row of the 2D array received as argument.

```
int add_first_row(int array[][MAX_COLS], int rows, int
cols)
{
   int sum = 0;
   for (int k = 0; k < cols; k++)
   {
      sum = sum + ____________________;
   }
   return sum;
}
```

# Arrays – Fixed Size is a Drawback

The size of an array *cannot* be changed after it is created.

You have to get the size right – *before* you define an array.

The compiler has to know the size to build it.
and a function must be told about the number
elements and possibly the capacity.

It cannot hold more than it's initial capacity.

*Later, we'll discuss vectors, which have variable size and some other programmer-friendly features lacking in arrays.*

# Summary 1

- Array is collection of "same type" data
- Indexed variables of array used just like any other simple variables
- for-loop "natural" way to traverse arrays
- Programmer responsible for staying "in bounds" of array
- Array parameter is "new" kind

# Summary 2

- Array elements stored sequentially
  - "Contiguous" portion of memory
  - Only address of 1st element is passed to functions
- Partially-filled arrays → more tracking
- Constant array parameters
  - Prevent modification of array contents
- Multidimensional arrays
  - Create "array of arrays"