Chapter Six: Arrays

Slides by Evan Gallagher

Chapter Goals

- To become familiar with using arrays to collect values
- To learn about common algorithms for processing arrays
- To write functions that receive and return arrays
- To be able to use two-dimensional arrays

Using Arrays

- when you need to work with a large number of values all together

- manage collections of data
- stored data is of the same type

Using Arrays

Think of a sequence of data:

32 54 67.5 29 35 80 115 44.5 100 65

(all of the same type, of course)
(storable as **`double`**s)

Using Arrays

32 54 67.5 29 35 80 115 44.5 100 65

Which is the largest in this set?

(You must look at every single value to decide.)

Using Arrays

32 54 67.5 29 35 80 115 44.5 100 65

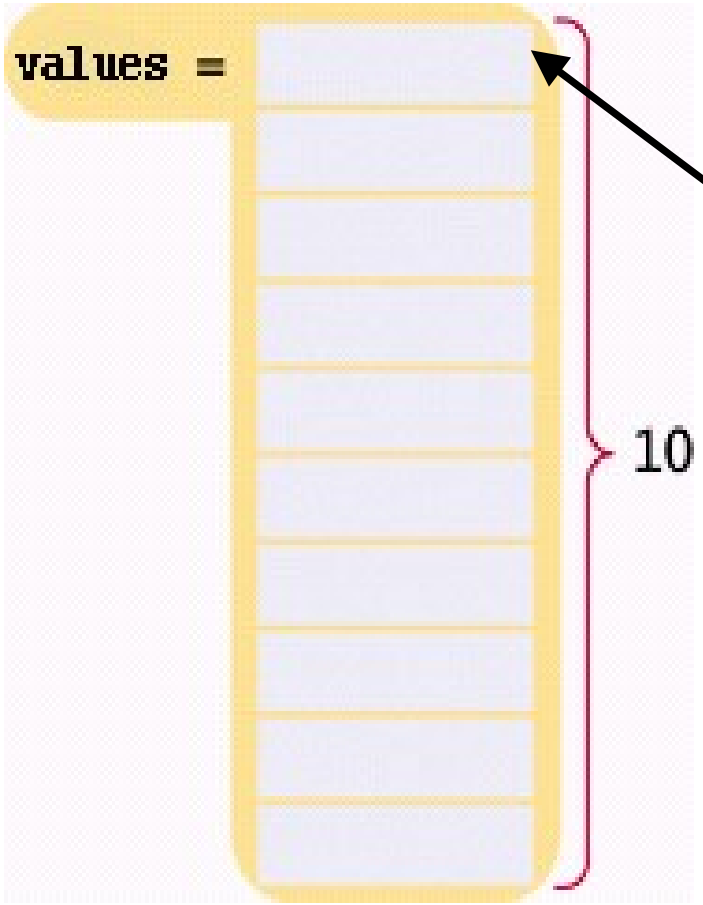
So you would create a variable for each, of course!

```
int n1, n2, n3, n4, n5, n6, n7, n8, n9, n10;
```

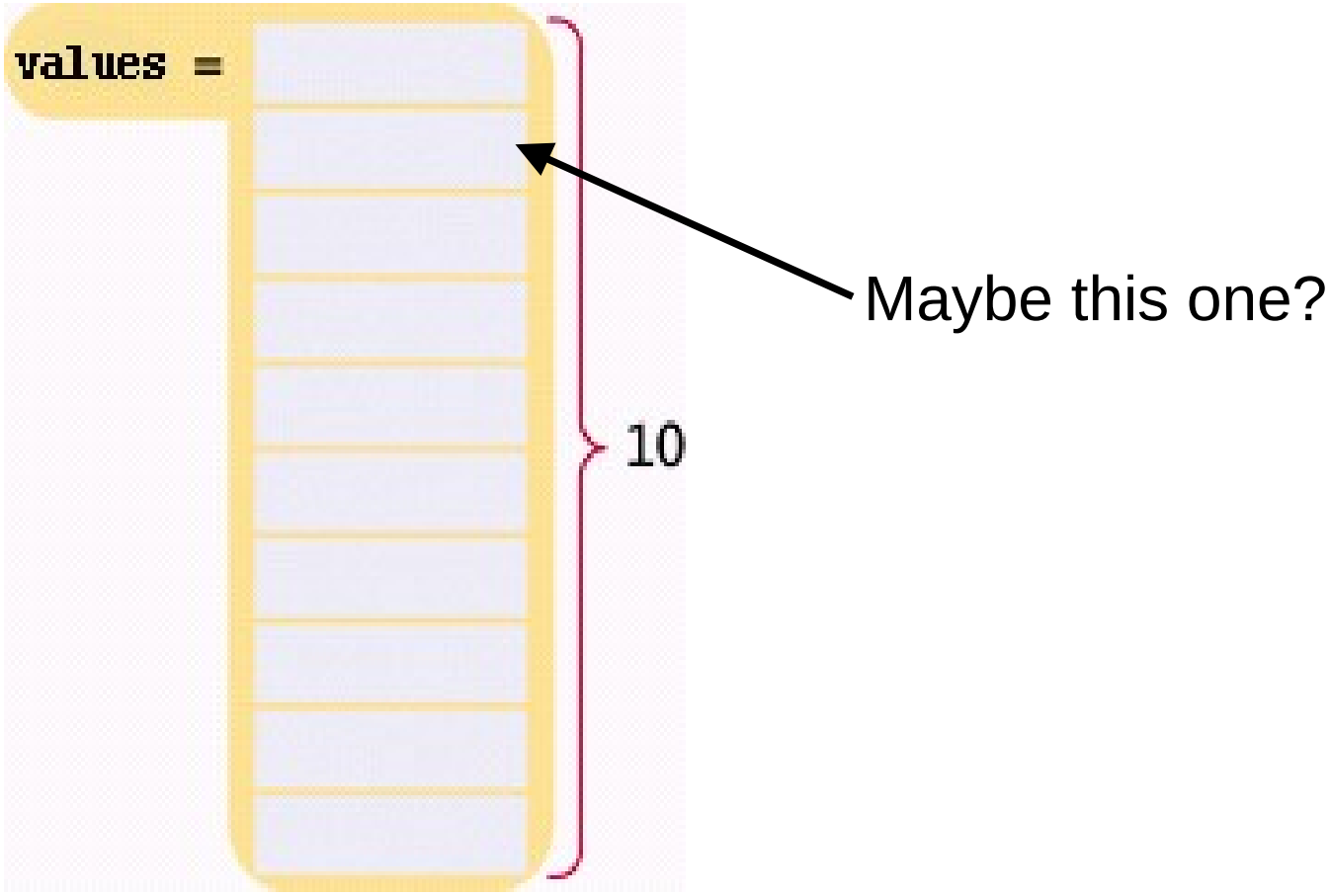
Then what???

Using Arrays

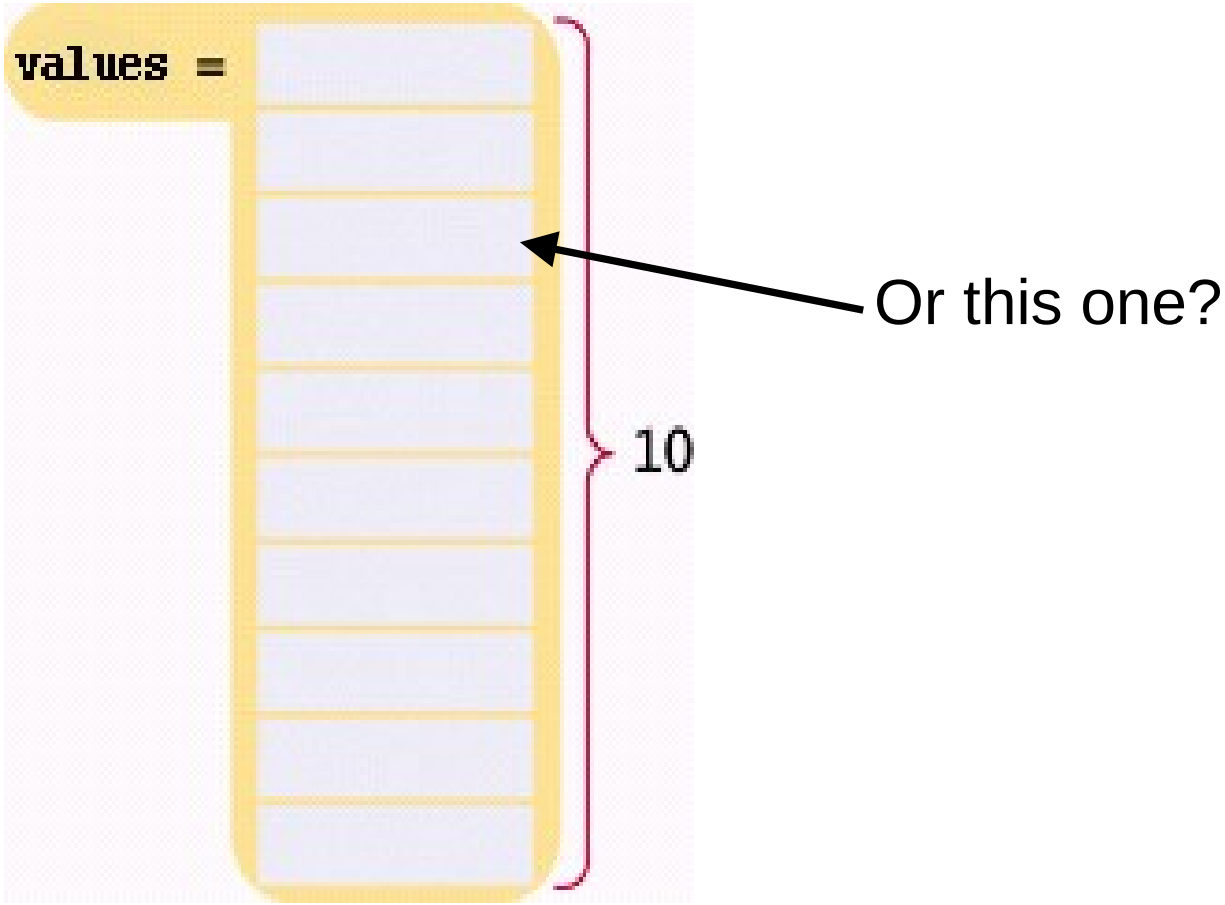
You can easily visit each element in an array,
checking and updating a variable holding the current maximum.



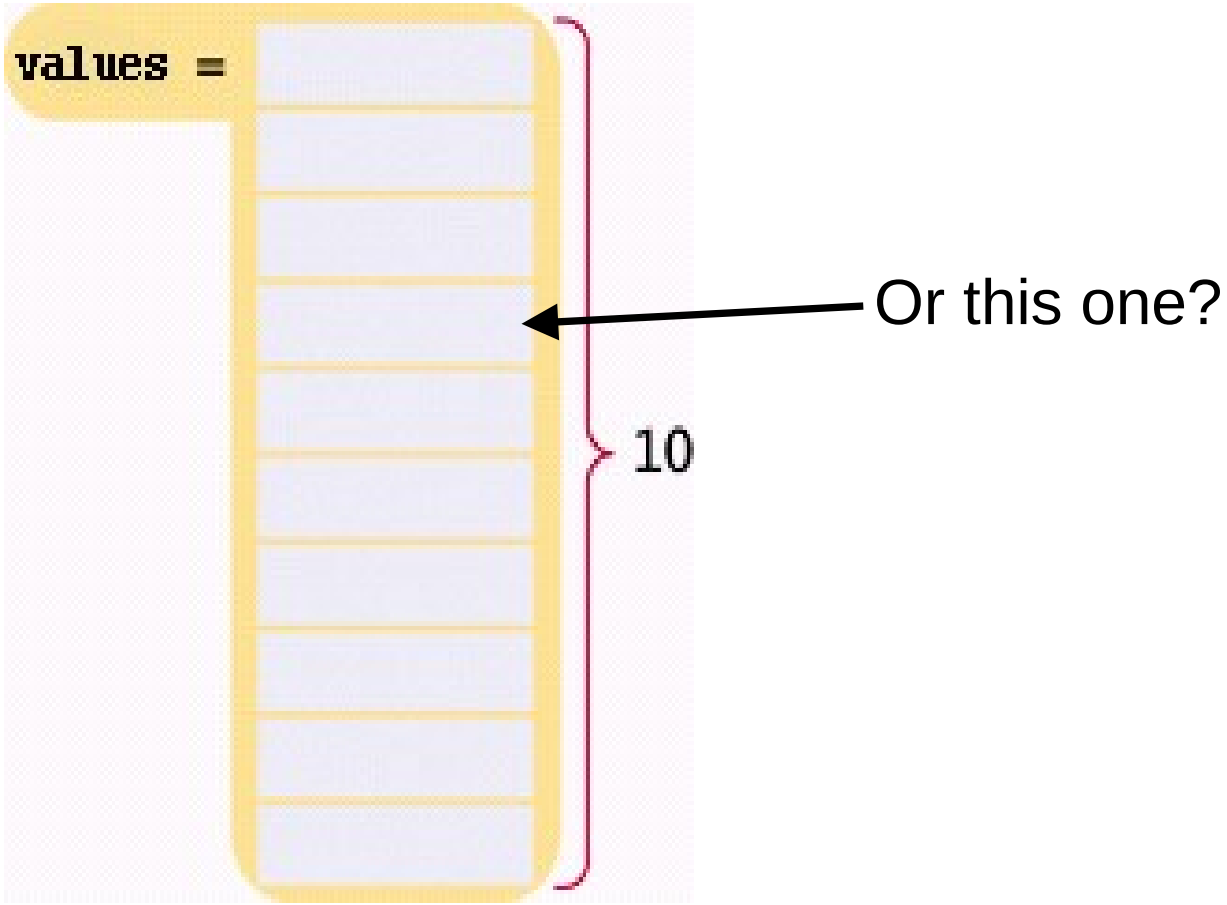
Using Arrays



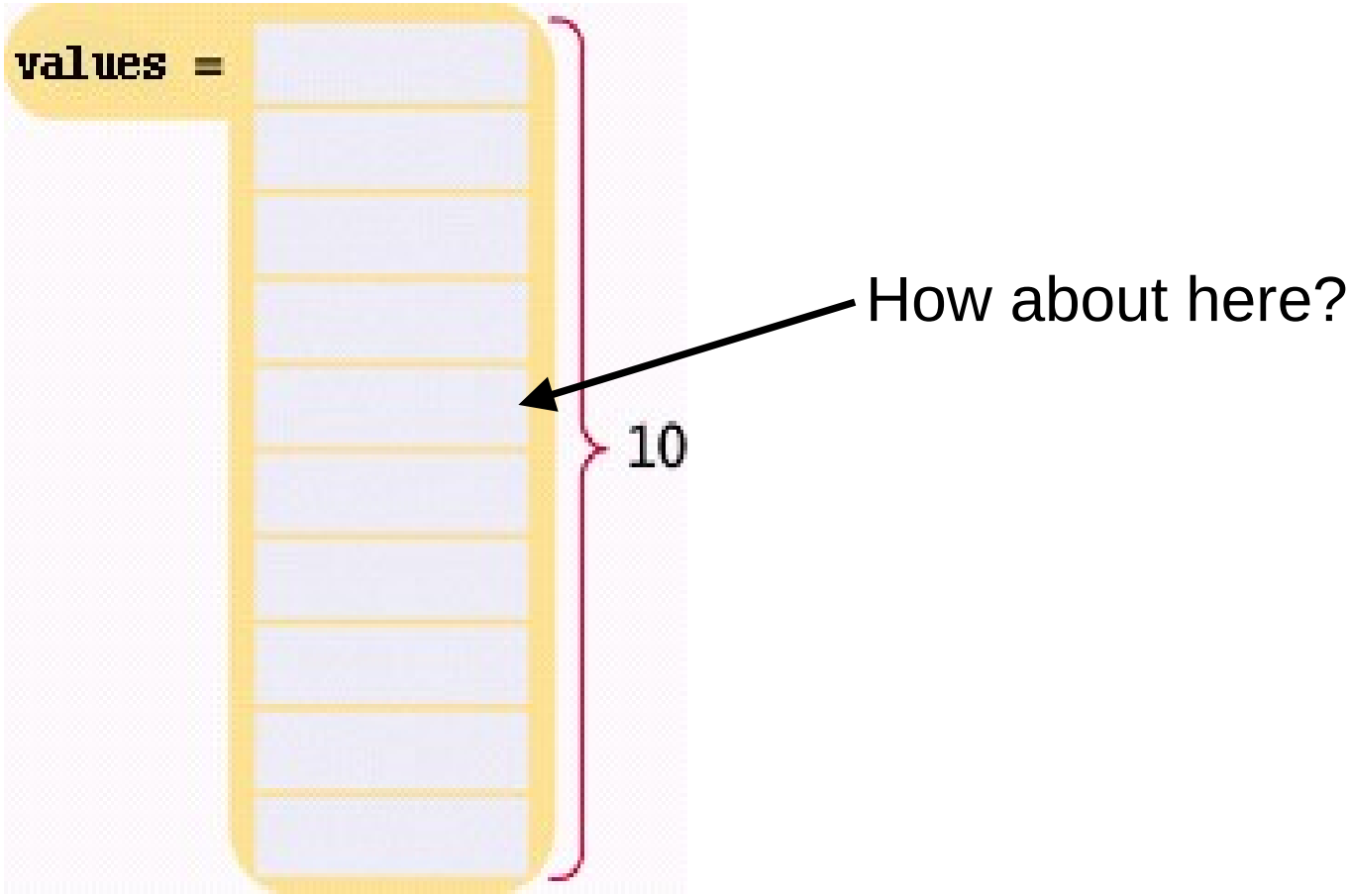
Using Arrays



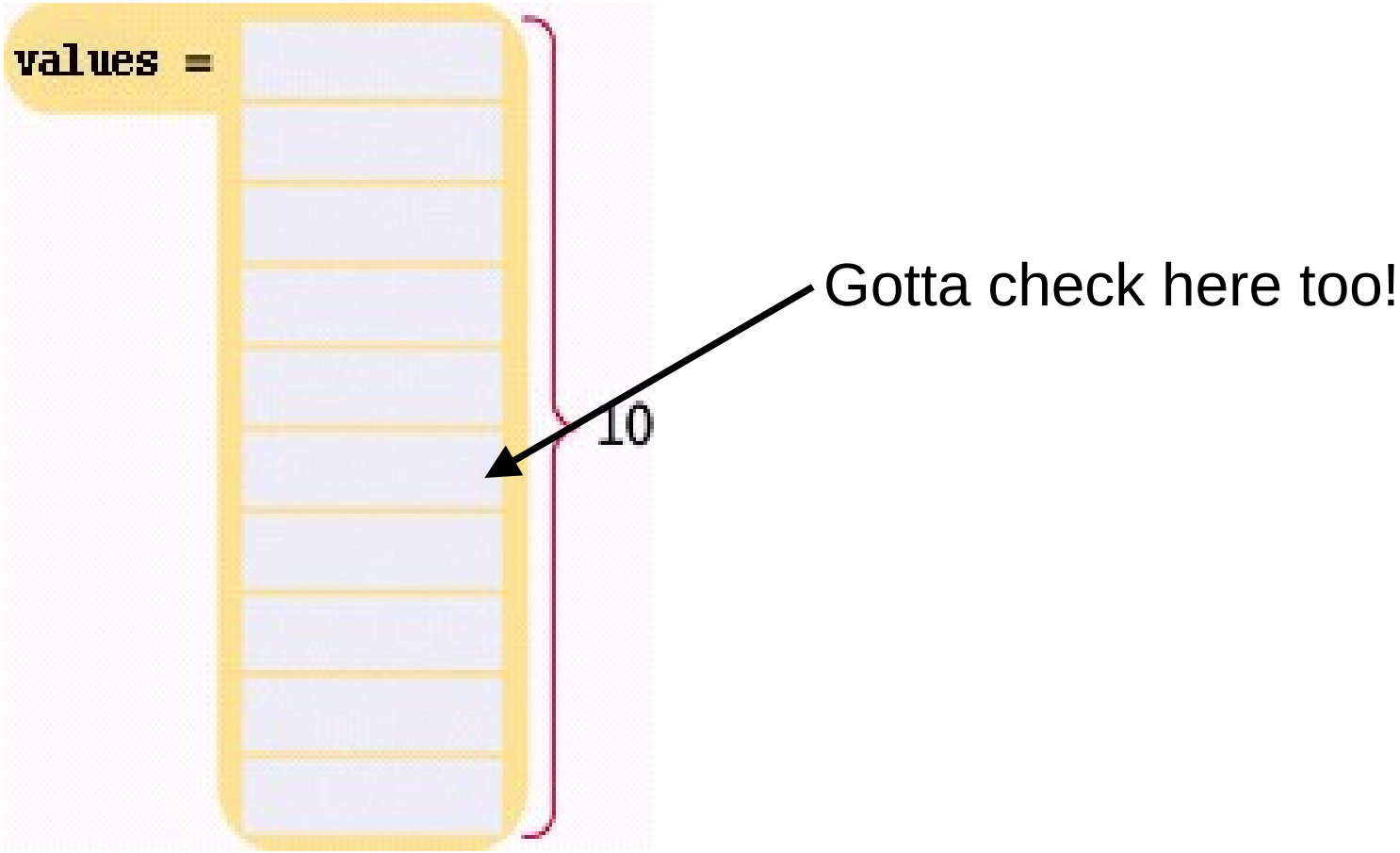
Using Arrays



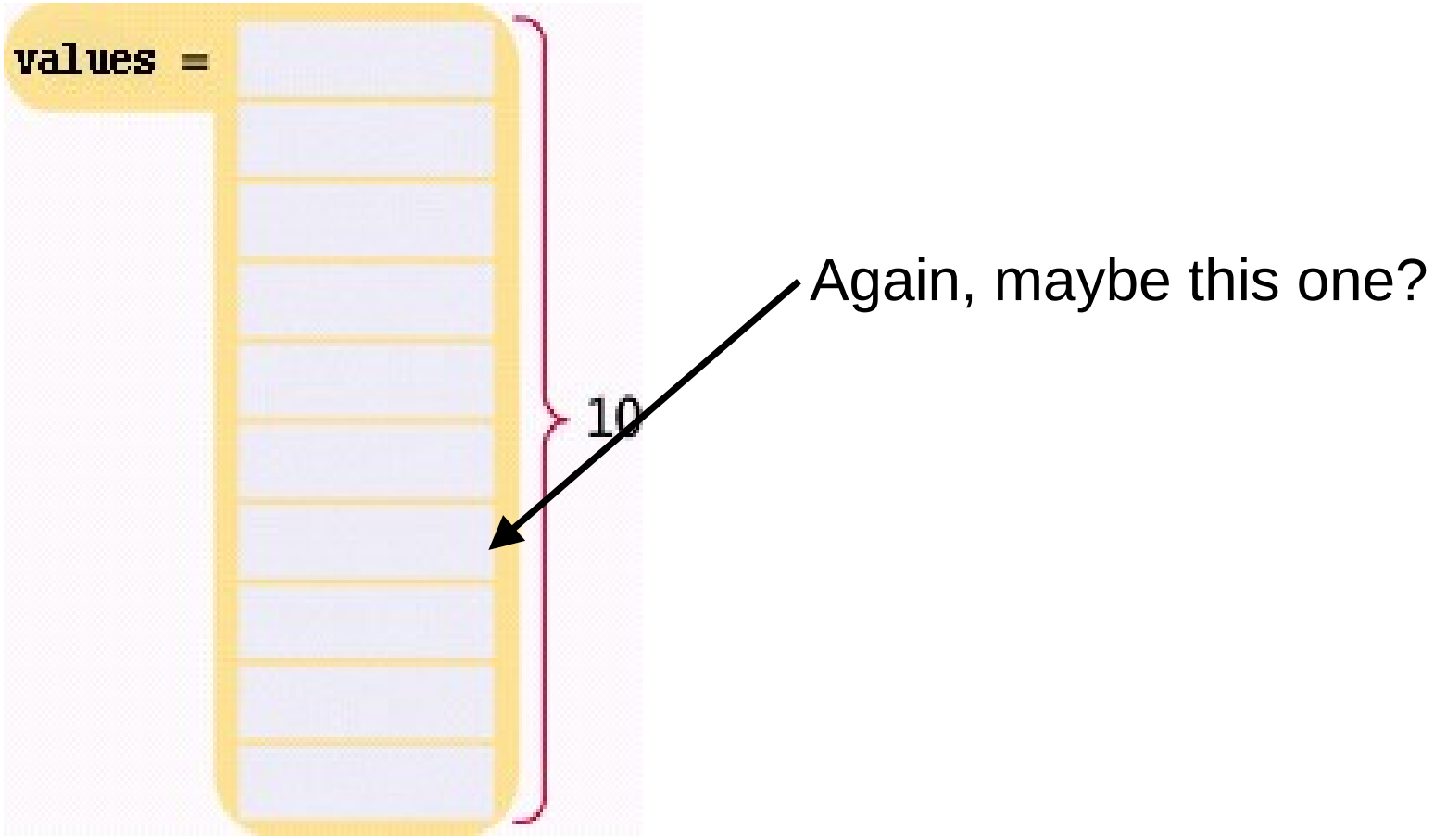
Using Arrays



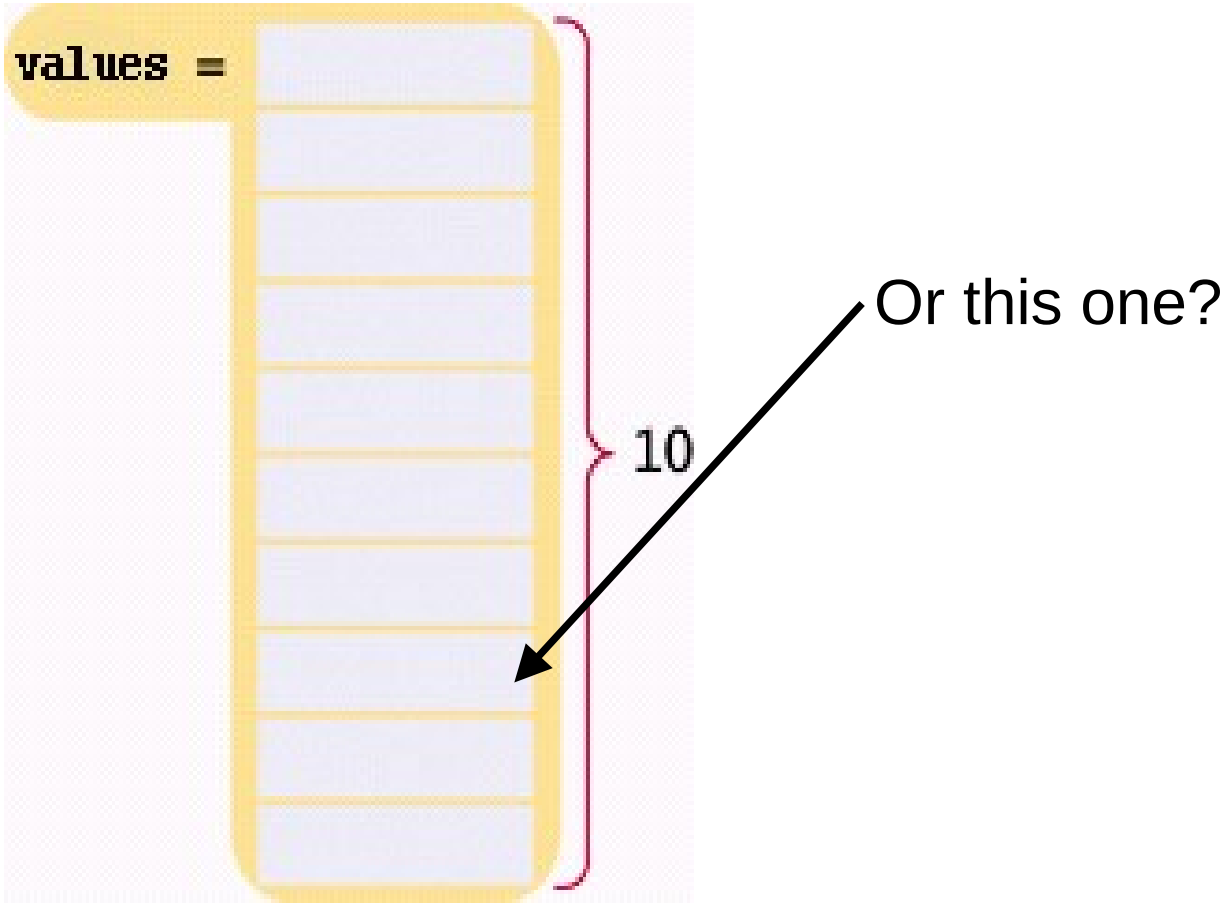
Using Arrays



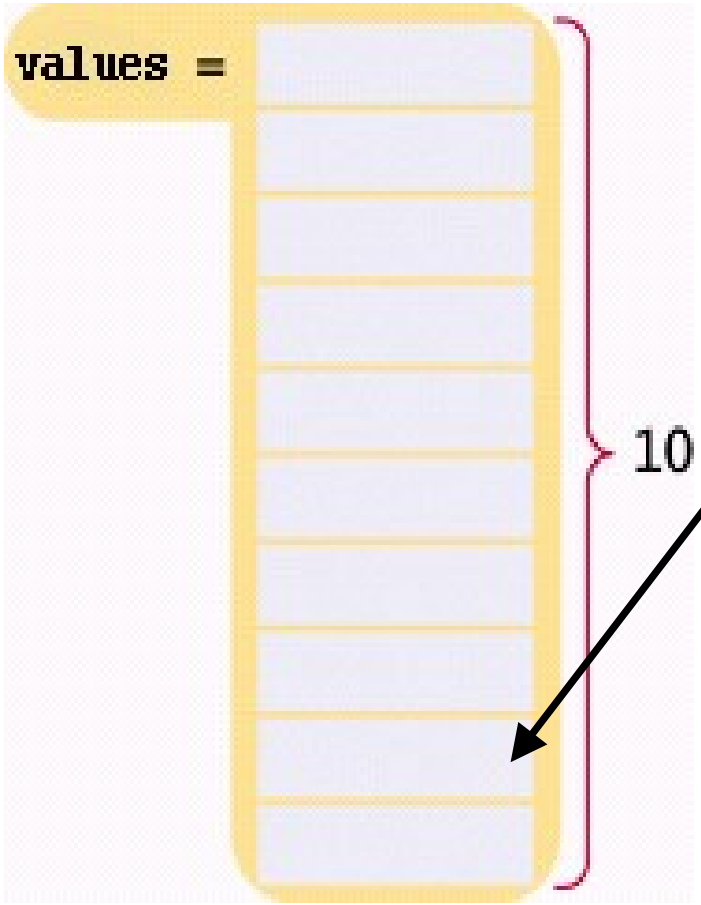
Using Arrays



Using Arrays



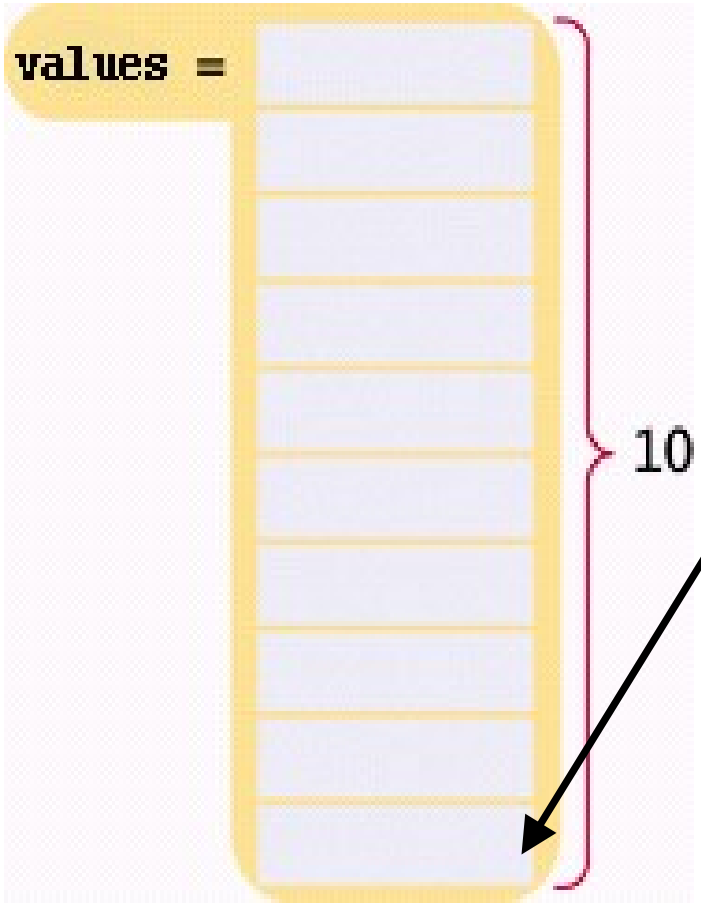
Using Arrays



Or this one?

Will this never end!

Using Arrays



Or the last one? ***Finally!***

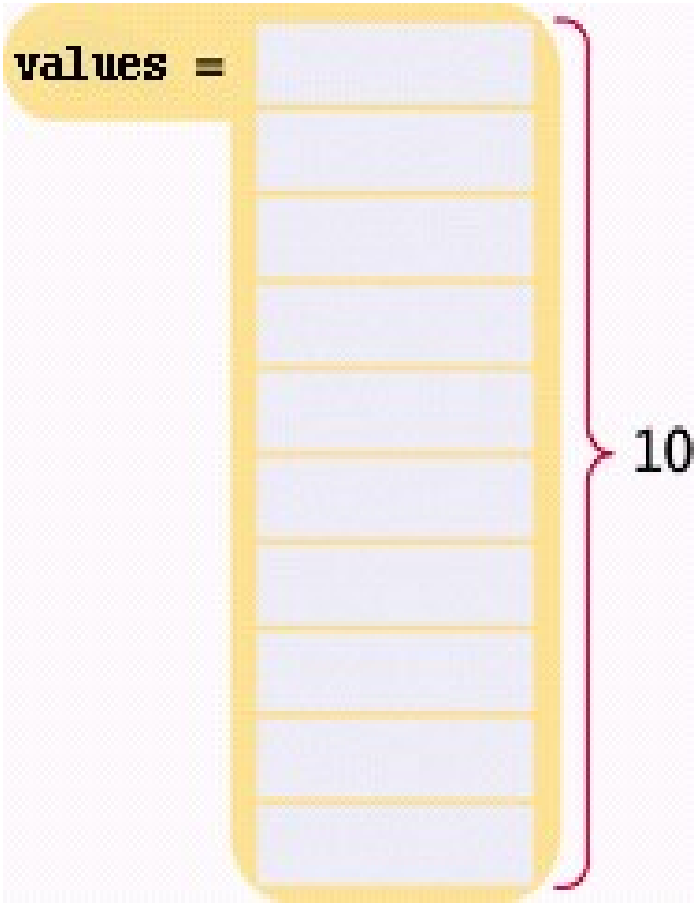
Using Arrays

That would have been impossible with ten separate variables!

```
int n1, n2, n3, n4, n5, n6, n7, n8, n9, n10;
```

And what if there needed to be another double in the set?

Defining Arrays



An “array of double”

Ten elements of double type can be stored under one name cs an array

```
double values[10];
```

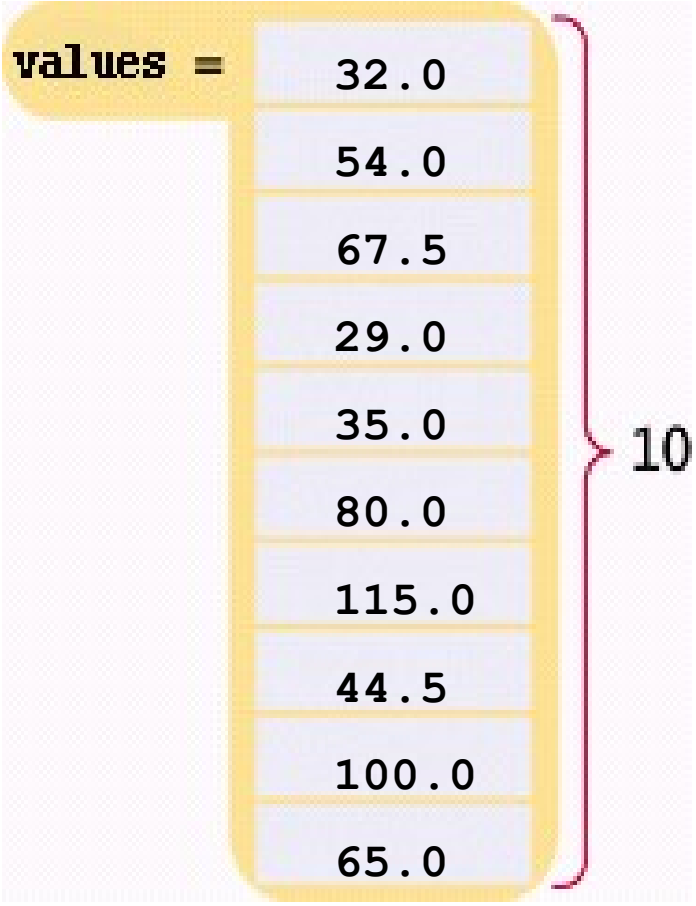
type of each element

quantity of elements – the “size” of the array, must be a constant

Defining Arrays with Initialization

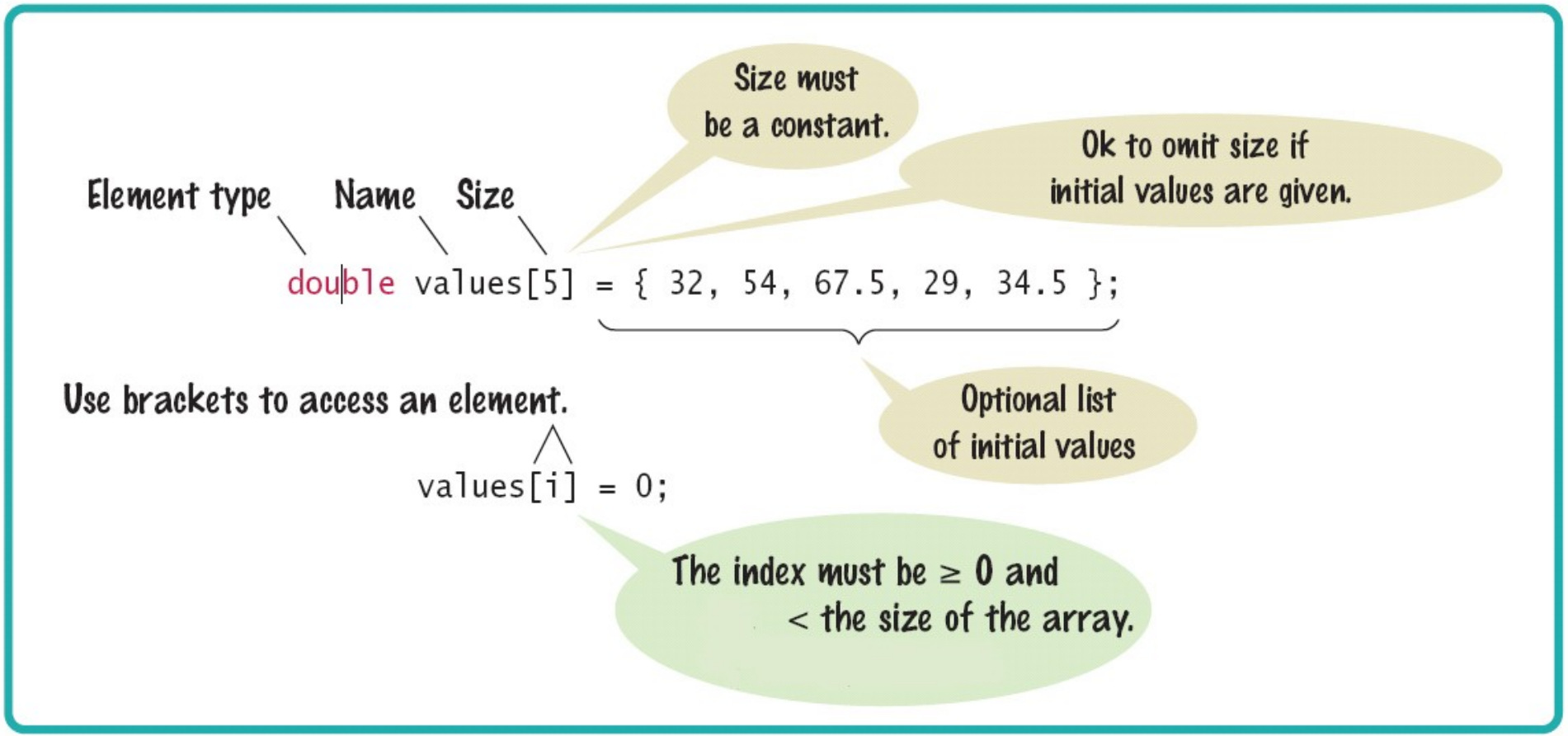
When you define an array, you can specify the initial values:

```
double values[] = { 32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65 };
```



Array Syntax

Defining an Array



Array Syntax

Table 1 Defining Arrays

`int numbers[10];`	An array of ten integers.
`const int SIZE = 10;` `int numbers[SIZE];`	It is a good idea to use a named constant for the size.
 `int size = 10;` `int numbers[size];`	**Caution:** In standard C++, the size must be a constant. This array definition will not work with all compilers.
`int squares[5] = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`int squares[] = { 0, 1, 4, 9, 16 };`	You can omit the array size if you supply initial values. The size is set to the number of initial values.
`int squares[5] = { 0, 1, 4 };`	If you supply fewer initial values than the size, the remaining values are set to 0. This array contains 0, 1, 4, 0, 0.
`string names[3];`	An array of three strings.

Accessing an Array Element

An array element can be used like any variable.

To access an array element, you use the notation:

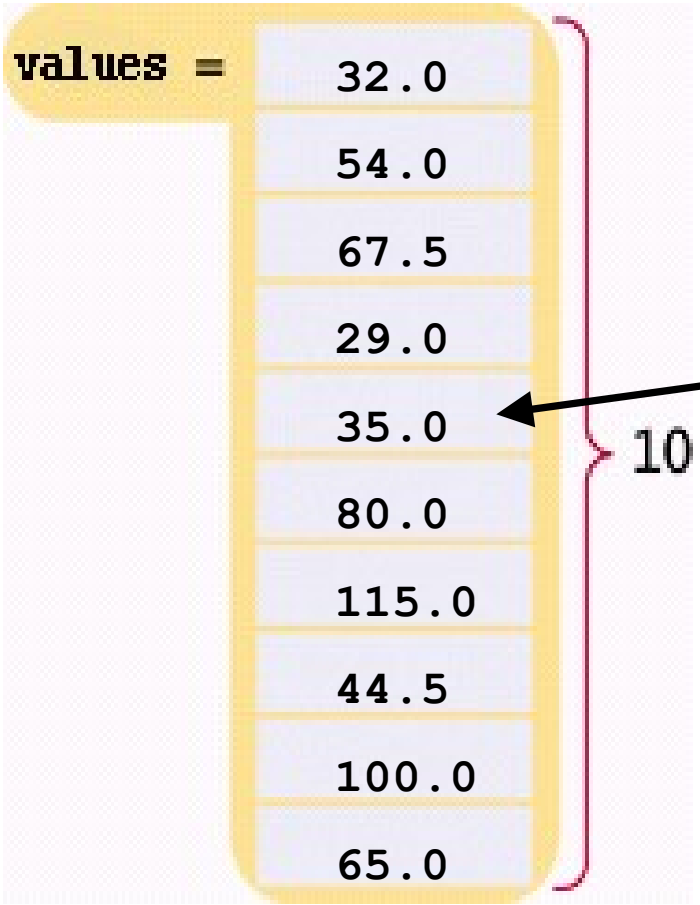
```
values[i]
```

where `i` is the *index*.

Accessing an Array Element

To access the element at index 4 using this notation: **`values[4]`**

4 is the *index*.

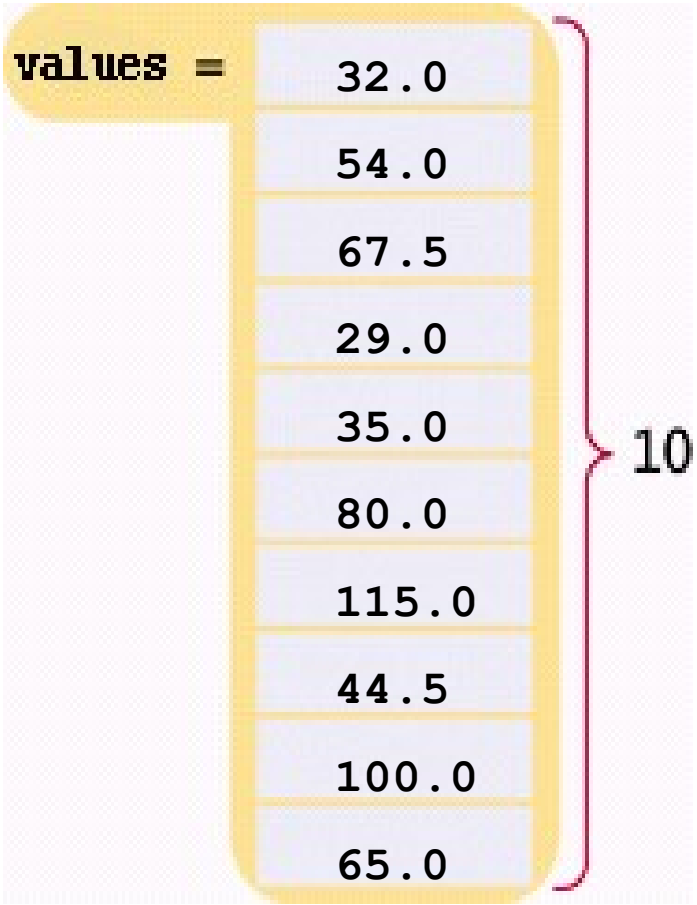



```
double values[10];
...
printf("%f\n", values[4]);
```

The output will be `35.0`.

Accessing an Array Element

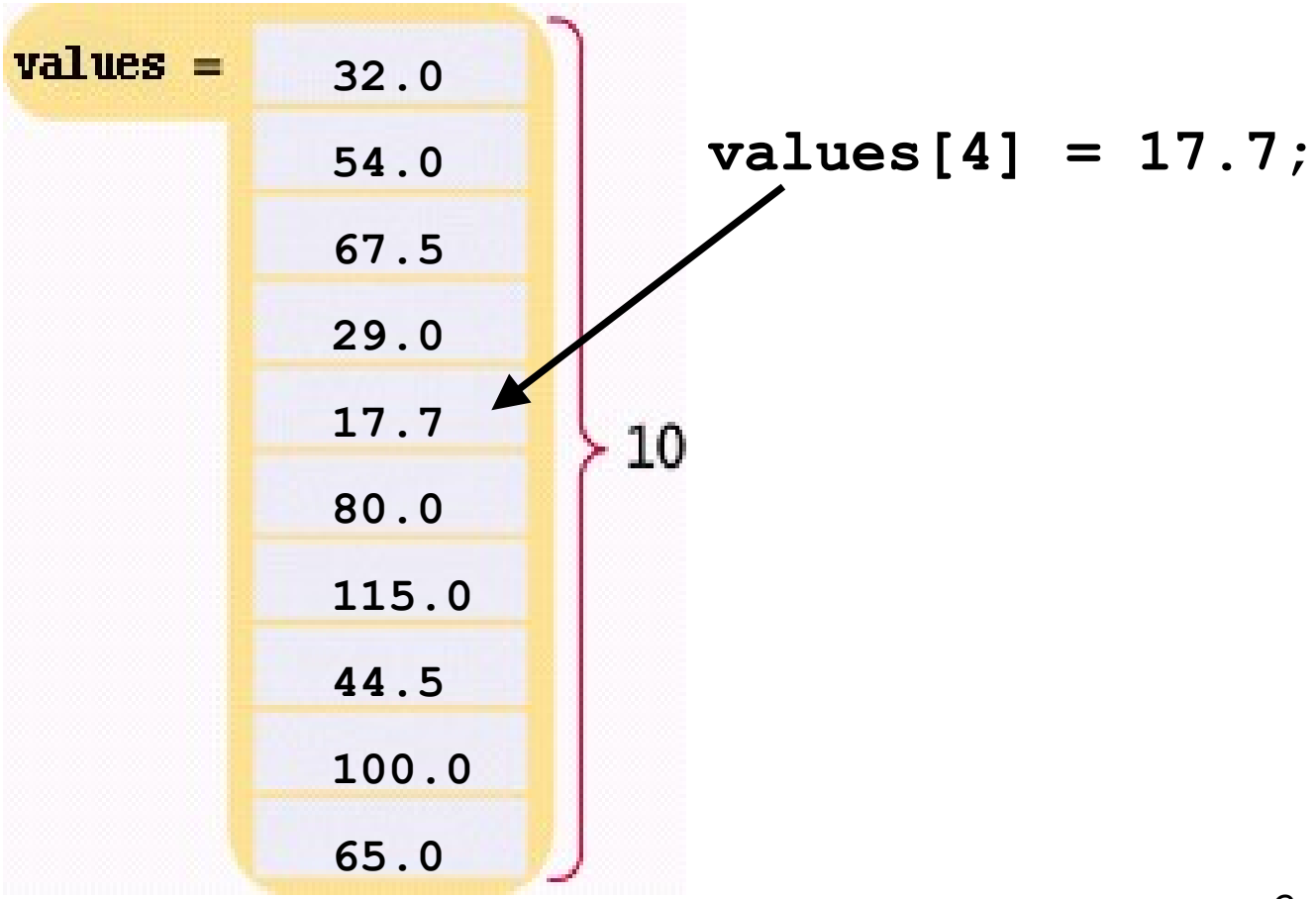
The same notation can be used to change the element.




```
values[4] = 17.7;
```

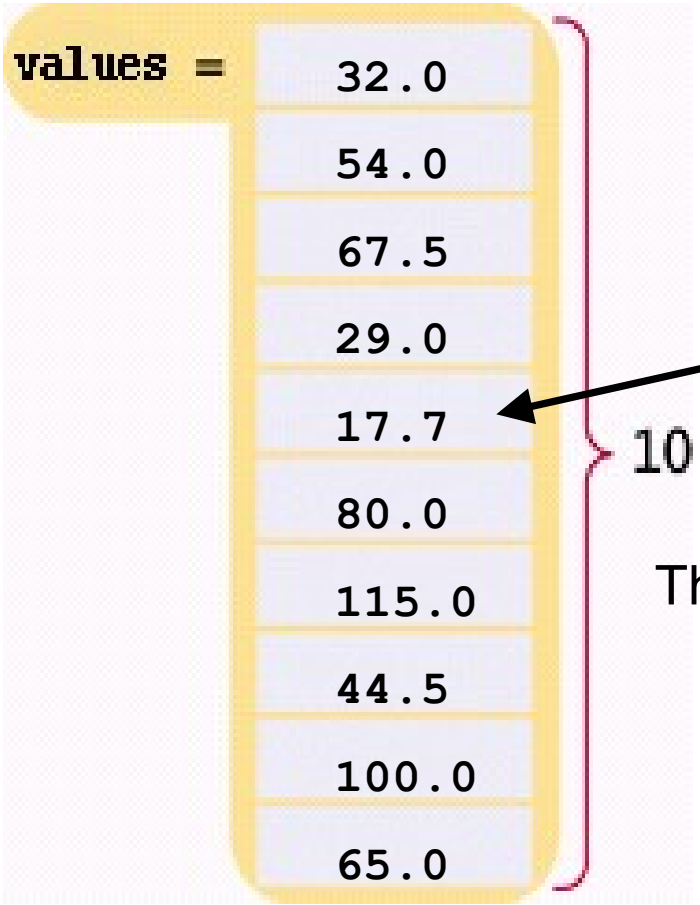
Accessing an Array Element

The same notation can be used to change the element.



Accessing an Array Element

The same notation can be used to change the element.

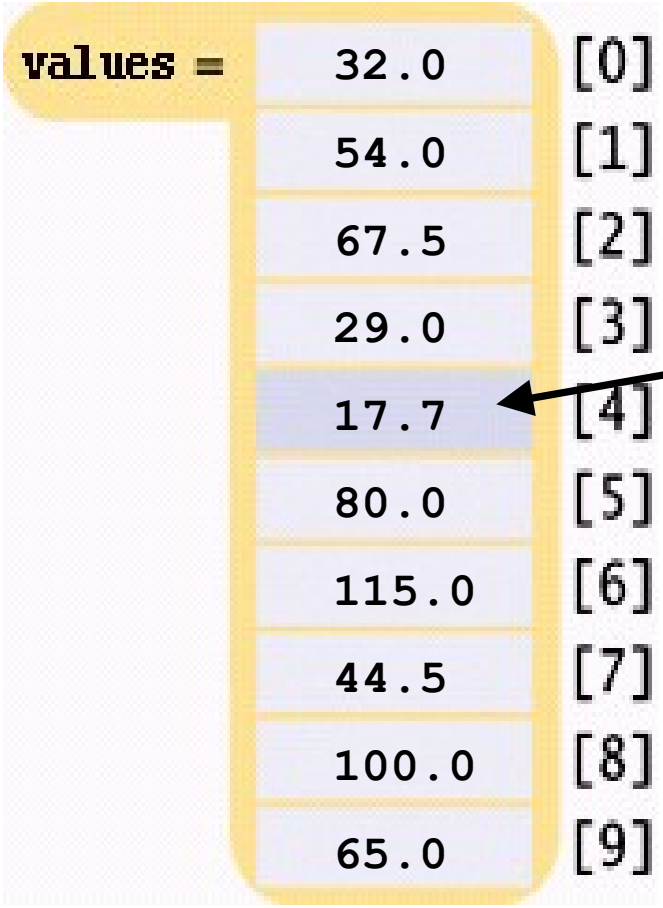



```
values[4] = 17.7;
printf("%f\n", values[4]);
```

The output will be `17.7`.

Accessing an Array Element

You might have thought those last two slides were wrong: `values[4]` is getting the data from the "fifth" element.




```
printf("%f\n", values[4]);
```

In C and most computer languages, indexing starts with **0**.

Accessing an Array Element

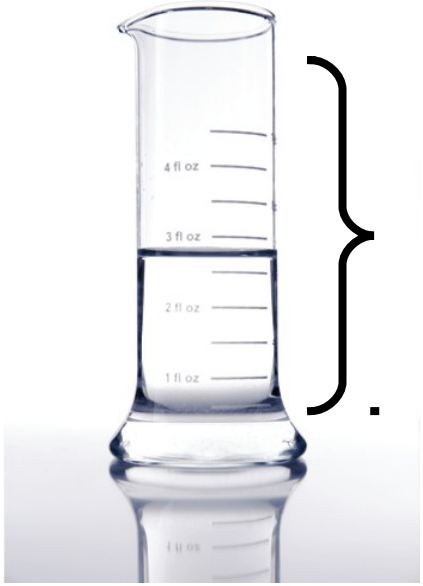
That is, the legal elements for the **`values`** array are:

`values[0]`, the ***first*** element
`values[1]`, the second element
`values[2]`, the third element
`values[3]`, the fourth element
`values[4]`, the fifth element
`...`
`values[9]`, the tenth ***and last legal*** element
recall: **`double values[10];`**

The index must be **`>= 0`** and **`<= 9`**.
0, 1, 2, 3, 4, 5, 6, 7, 8, 9 is 10 numbers.

Partially-Filled Arrays

Suppose an array can hold 10 elements:

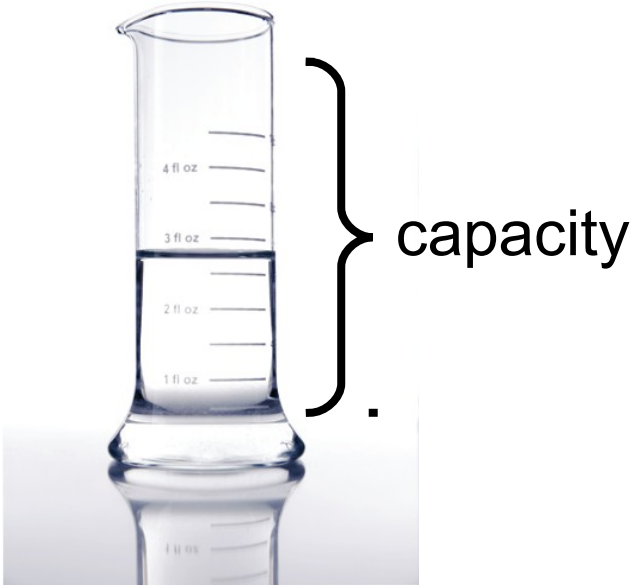


Does it always?
Just look at that beaker.
Guess not!

Partially-Filled Arrays – Capacity

How many elements, at most, can an array hold?

We call this quantity the *capacity*.



Partially-Filled Arrays – Capacity

For example, we may decide for a particular problem that there are usually ten or 11 values, but never more than 100.

We would set the capacity with a **`const`**:

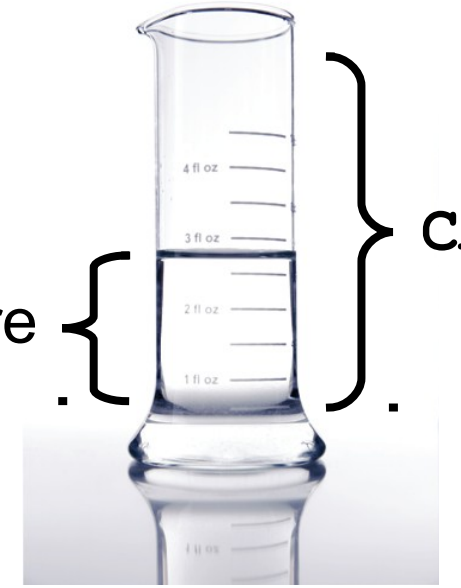
```
const int CAPACITY = 100;
double values[CAPACITY];
```

Partially-Filled Arrays

Arrays will usually hold less than **`CAPACITY`** elements.

We call this kind of array a *partially filled array*:

only partially filled to here

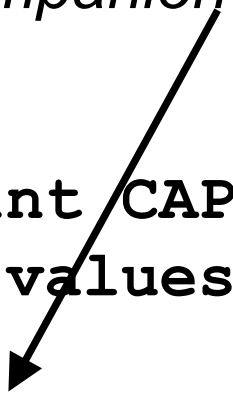


`CAPACITY`

Partially-Filled Arrays – Companion Variable for Size

But how many actual elements are there in a partially filled array?

We will use a *companion variable* to hold that amount:

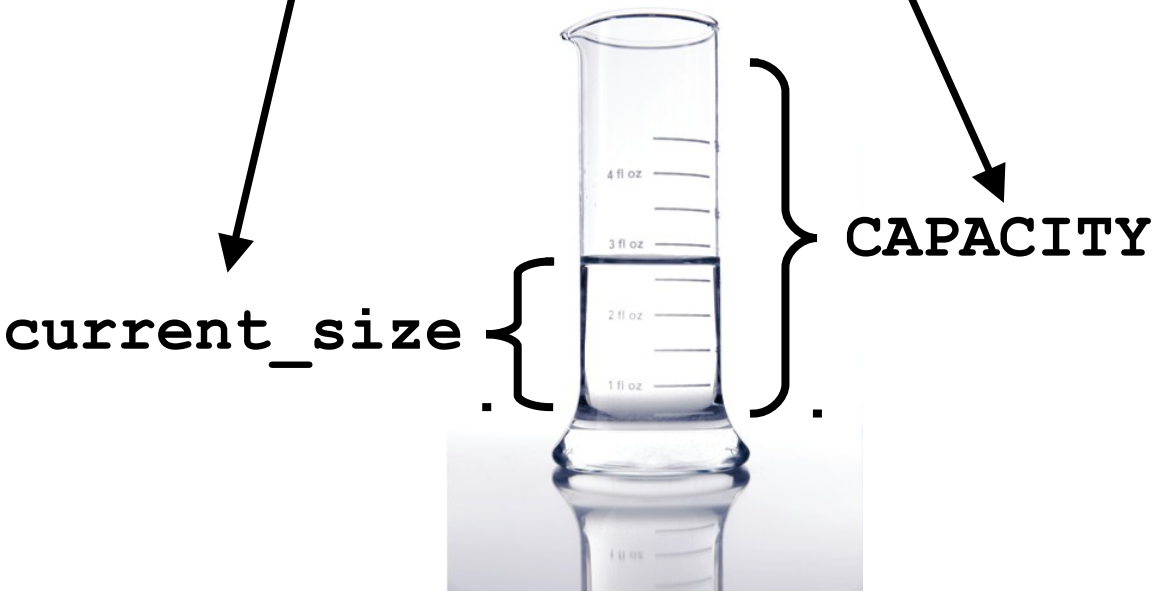


```
const int CAPACITY = 100;
double values[CAPACITY];

int current_size = 0; // array is empty
```

Suppose we add four elements to the array?

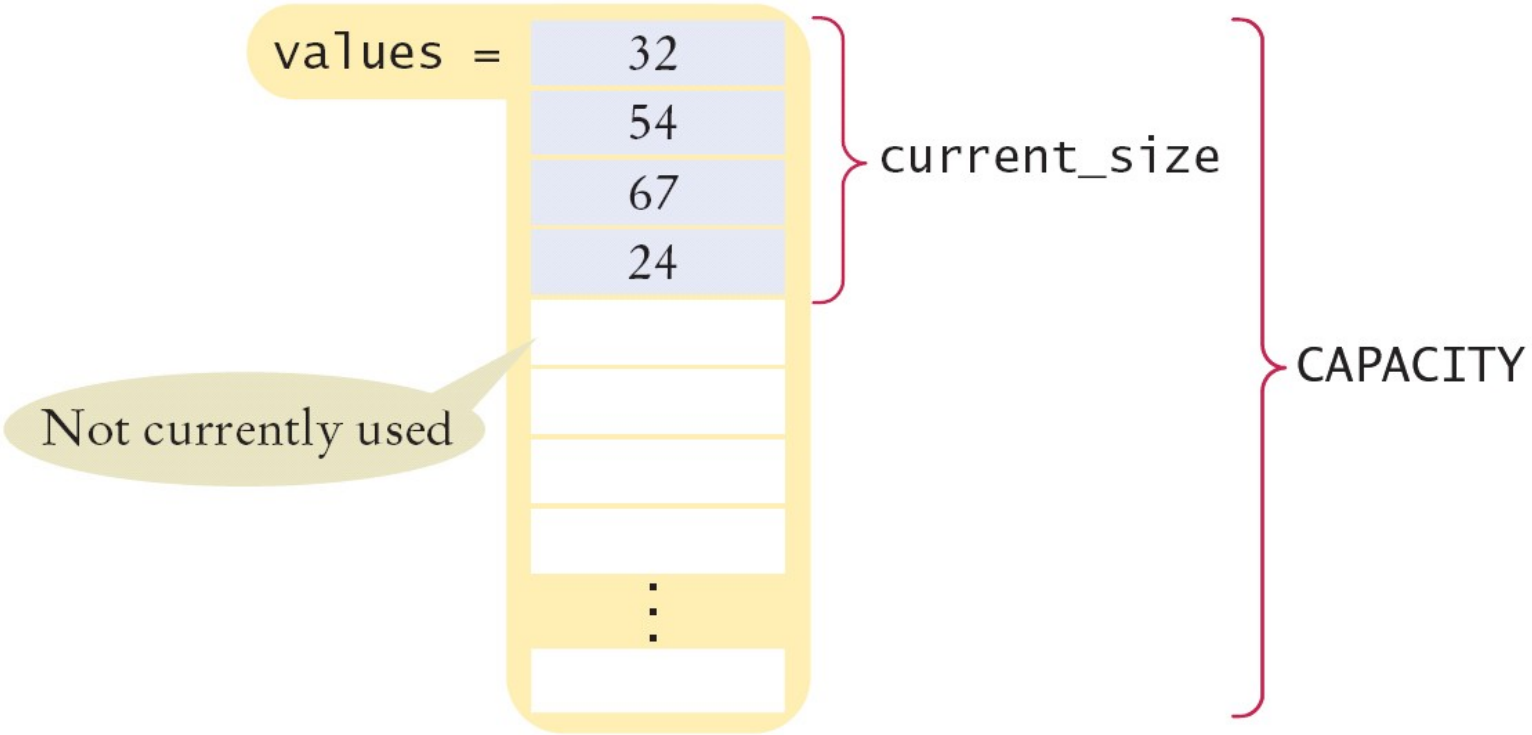
Partially-Filled Arrays – Companion Variable for Size

```
const int CAPACITY = 100;
double values[CAPACITY];

current_size = 4; // array now holds 4
```



Partially-Filled Arrays – Companion Variable for Size

```
const int CAPACITY = 100;
double values[CAPACITY];

current_size = 4; // array now holds 4
```



Partially-Filled Arrays – Capacity

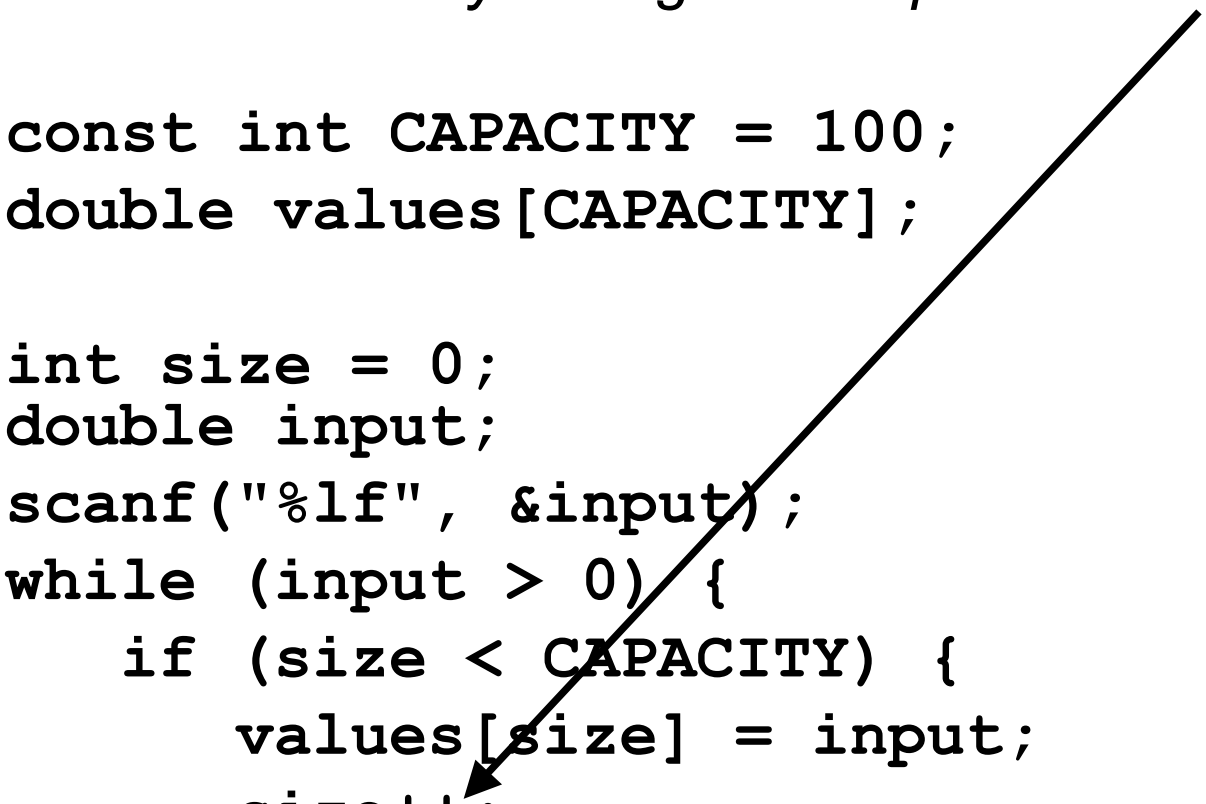
The following loop fills an array with user input.

Each time the size of the array changes we update this variable:


```
const int CAPACITY = 100;
double values[CAPACITY];

int size = 0;
double input;
scanf("%lf", &input);
while (input > 0) {
   if (size < CAPACITY) {
      values[size] = input;
      size++;
   }
   scanf("%lf", &input);
}
```

Partially-Filled Arrays – Capacity

The following loop fills an array with user input.
Each time the size of the array changes we update this variable:



```
const int CAPACITY = 100;
double values[CAPACITY];

int size = 0;
double input;
scanf("%lf", &input);
while (input > 0) {
   if (size < CAPACITY) {
      values[size] = input;
      size++;
   }
   scanf("%lf", &input);
}
```

Partially-Filled Arrays – Capacity

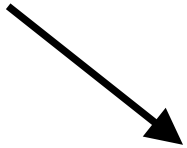
When the loop ends, the companion variable `size` has the number of elements in the array.

```
const int CAPACITY = 100;
double values[CAPACITY];

int size = 0;
double input;
scanf("%lf", &x);
while (input > 0) {
   if (size < CAPACITY) {
      values[size] = input;
      size++;
   }
   scanf("%lf", &input);
}
```

Partially-Filled Arrays – Visiting All Elements

How would you print the elements in a partially filled array?

By using the **`current_size`** companion variable.



```
for (int i = 0; i < current_size; i++) {
   printf("%f\n", values[i]);
}
```

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A `for` loop's variable is best:

```
for (int i = 0; i < current_size; i++) {
   printf("%f\n", values[i]);
}
```

When `i` is `0`,

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A `for` loop's variable is best:

```
for (int i = 0; i < current_size; i++) {
   printf("%f\n", values[i]);
}
```

When `i` is `0`, `values[i]` is `values[0]`, the first element.

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A `for` loop's variable is best:

```
for (int i = 0; i < current_size; i++) {
   printf("%f\n", values[i]);
}
```

When `i` is `0`, `values[i]` is `values[0]`, the first element.

When `i` is `1`,

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A `for` loop's variable is best:

```
for (int i = 0; i < current_size; i++) {
   printf("%f\n", values[i]);
}
```

When `i` is `0`, `values[i]` is `values[0]`, the first element.

When `i` is `1`, `values[i]` is `values[1]`, the second element.

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A `for` loop's variable is best:

```
for (int i = 0; i < current_size; i++) {
   printf("%f\n", values[i]);
}
```

When `i` is `0`, `values[i]` is `values[0]`, the first element.

When `i` is `1`, `values[i]` is `values[1]`, the second element.

When `i` is `2`,

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A `for` loop's variable is best:

```
for (int i = 0; i < current_size; i++) {
   printf("%f\n", values[i]);
}
```

When `i` is `0`, `values[i]` is `values[0]`, the first element.

When `i` is `1`, `values[i]` is `values[1]`, the second element.

When `i` is `2`, `values[i]` is `values[2]`, the third element.

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A `for` loop's variable is best:

```
for (int i = 0; i < current_size; i++) {
   printf("%f\n", values[i]);
}
```

When `i` is `0`, `values[i]` is `values[0]`, the first element.

When `i` is `1`, `values[i]` is `values[1]`, the second element.

When `i` is `2`, `values[i]` is `values[2]`, the third element.

...

When `i` is `9`,

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A **`for`** loop's variable is best:

```
for (int i = 0; i < current_size; i++) {
   printf("%f\n", values[i]);
}
```

When **`i`** is **`0`**, **`values[i]`** is **`values[0]`**, the first element.

When **`i`** is **`1`**, **`values[i]`** is **`values[1]`**, the second element.

When **`i`** is **`2`**, **`values[i]`** is **`values[2]`**, the third element.

`...`

When **`i`** is **`9`**, **`values[i]`** is **`values[9]`**,
the ***last legal*** element.

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A **`for`** loop's variable is best:

```
for (int i = 0; i < current_size; i++) {
   printf("%f\n", values[i]);
}
```

Note that the loop condition is that the index is

less than **`current_size`**

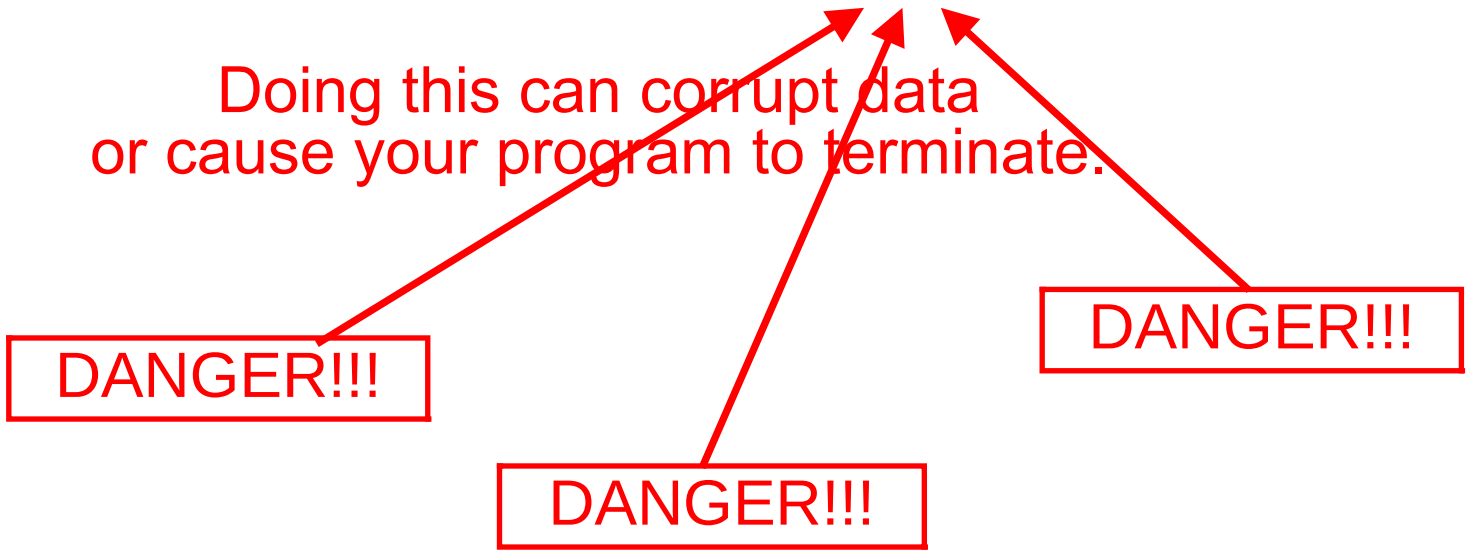
because there is no element corresponding to data[10].

But **`current_size`** (10) ***is*** the number of elements we want to visit.

Illegally Accessing an Array Element – *Bounds Error*

A *bounds* error occurs when you access an element outside the legal set of indices:

```
printf("%f", values[10]);
```



Use Arrays for Sequences of Related Values

Recall that the type of every element must be the same.

That implies that the “meaning” of each stored value is the same.

```
int scores[NUMBER_OF_SCORES];
```

Clearly the meaning of each element is a score.

Use Arrays for Sequences of Related Values

But an array could be used improperly:

```
double personal_data[3];
personal_data[0] = age;
personal_data[1] = bank_account;
personal_data[2] = shoe_size;
```

Clearly these **doubles** do *not* have the same meaning!

Use Arrays for Sequences of Related Values

But worse:

```
personal_data[   ] = new_shoe_size;
```

Use Arrays for Sequences of Related Values

But worse:

```
personal_data[ ? ] = new_shoe_size;
```

Oh dear!

Which position was I using for the shoe size?

Use Arrays for Sequences of Related Values

Arrays should be used when

the meaning of each element is the same.

Common Array Algorithms

There are many typical things that are done with sequences of values.

There many common algorithms for processing values stored in arrays.

Common Algorithms – Filling

This loop fills an array with zeros:

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

Common Algorithms – Filling

Here, we fill the array with squares (0, 1, 4, 9, 16, ...).

Note that the element with index 0 will contain 0^2,
the element with index 1 will contain 1^2, and so on.

```
for (int i = 0; i < size of squares; i++) {
   squares[i] = i * i;
}
```

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`**?

Common Algorithms – Copying

Let's try what seems right and easy…

```
squares = lucky_numbers;
```

…and wrong!

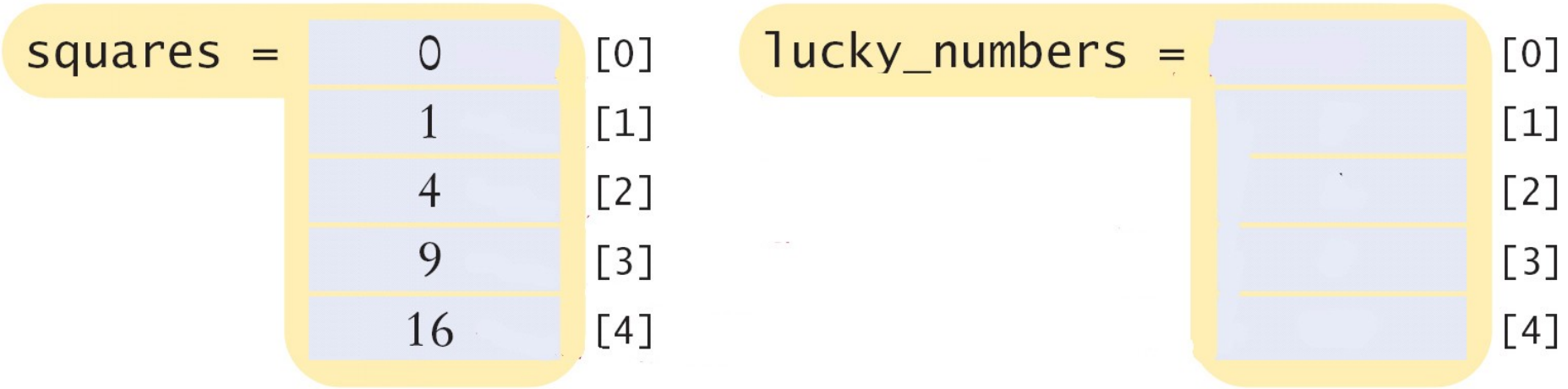
You cannot assign arrays!

You will have to do your own work.

Common Algorithms – Copying

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

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

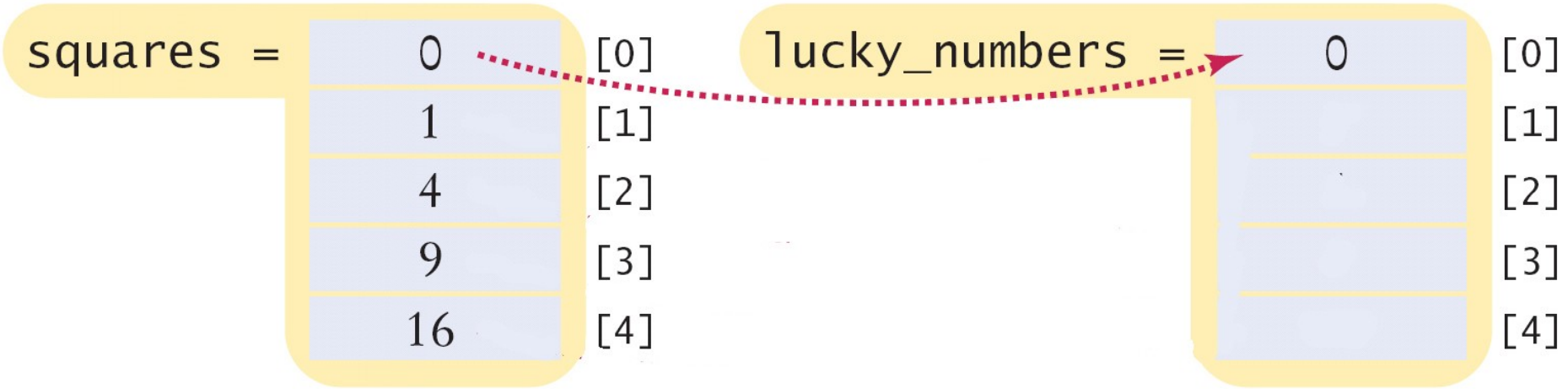
when `i` is 0



Common Algorithms – Copying

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

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

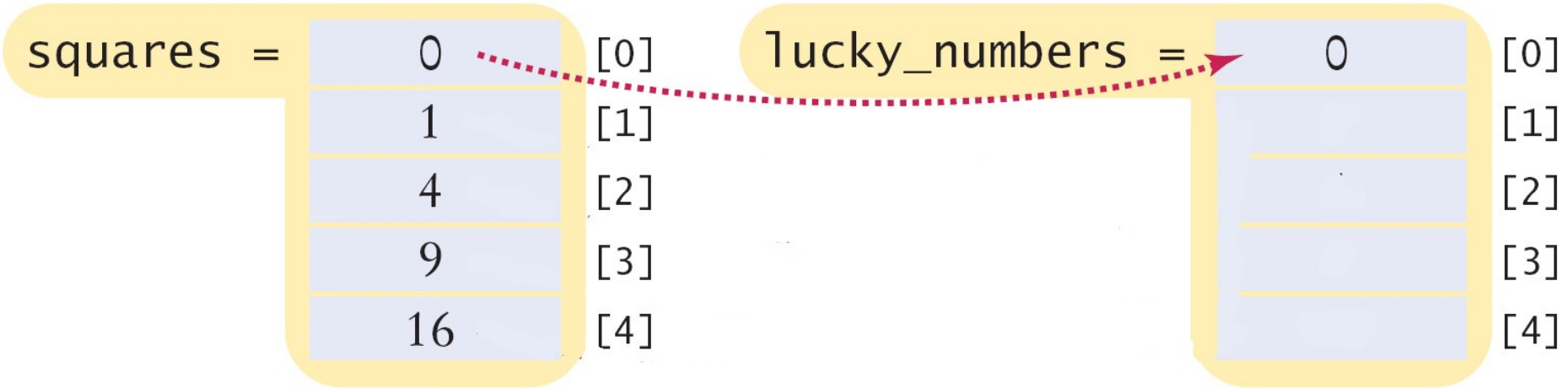
when `i` is `0`



Common Algorithms – Copying

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

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

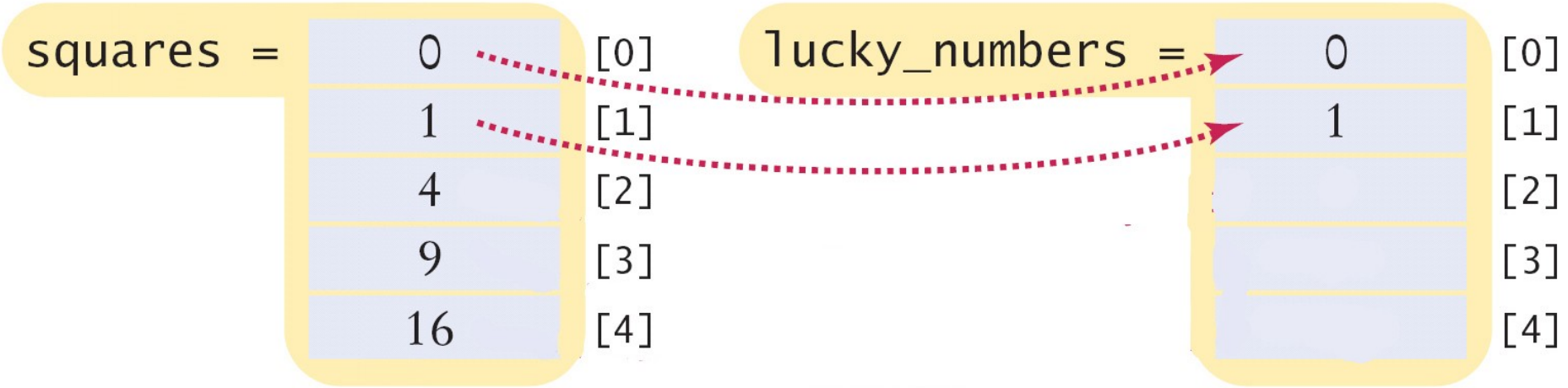
when `i` is `1`



Common Algorithms – Copying

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

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

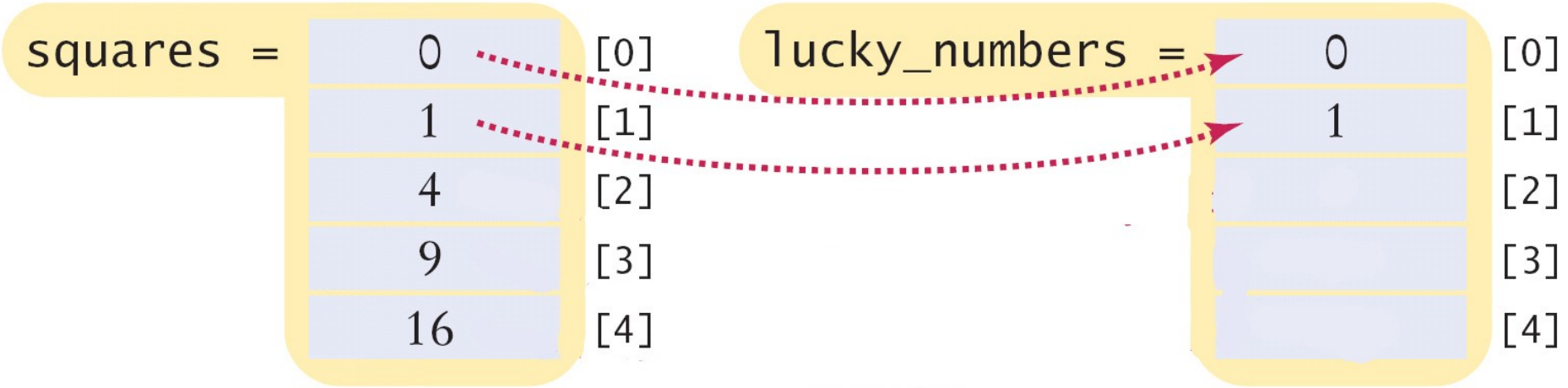
when `i` is `1`



Common Algorithms – Copying

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

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

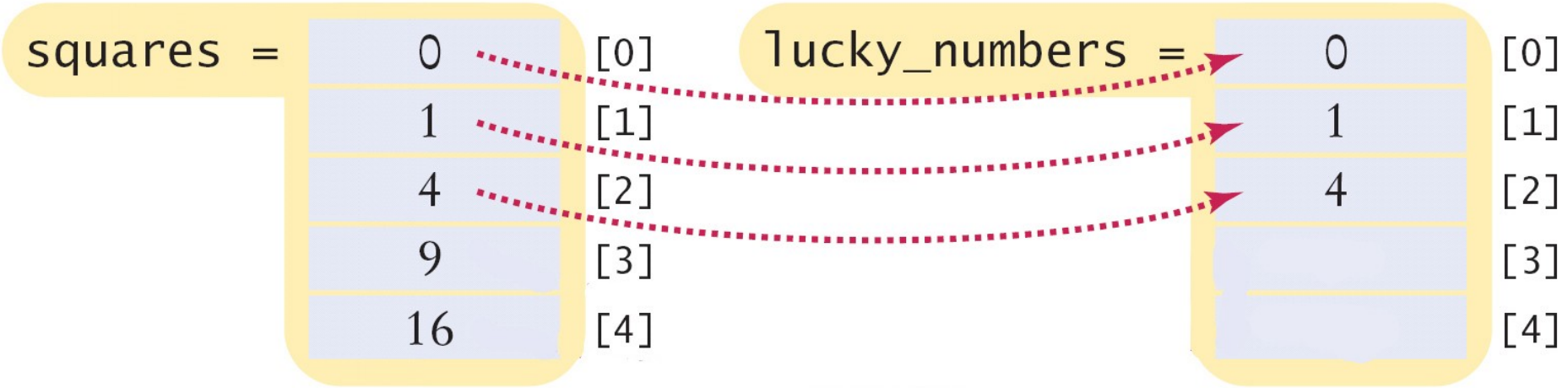
when `i` is 2



Common Algorithms – Copying

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

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

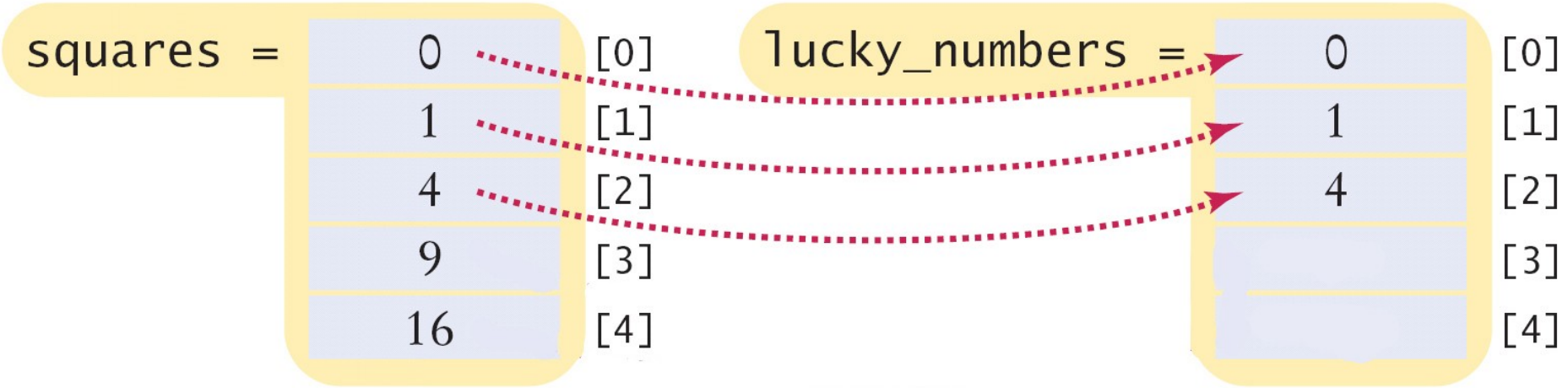
when `i` is 2



Common Algorithms – Copying

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

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

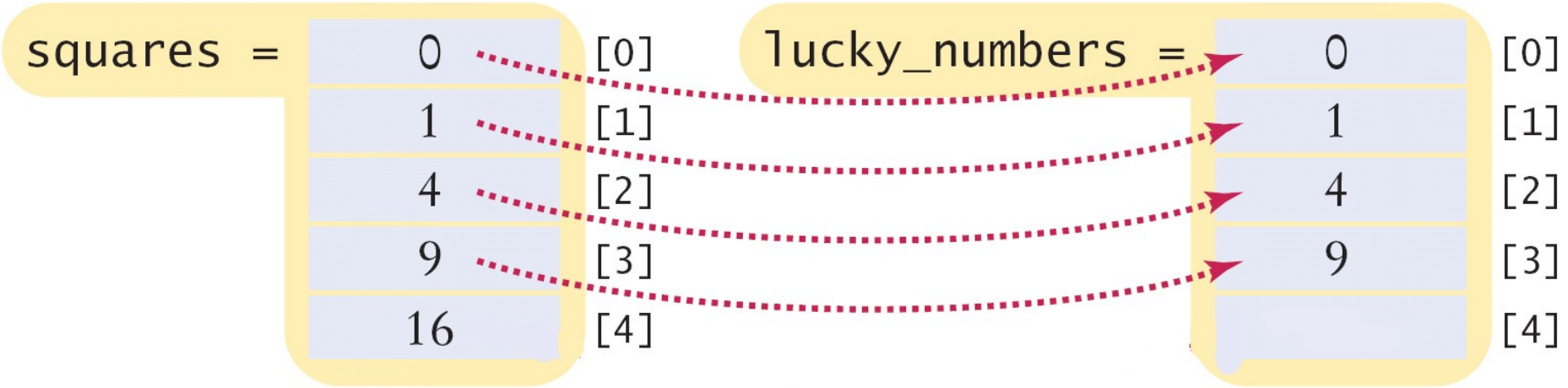
when `i` is 3



Common Algorithms – Copying

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

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

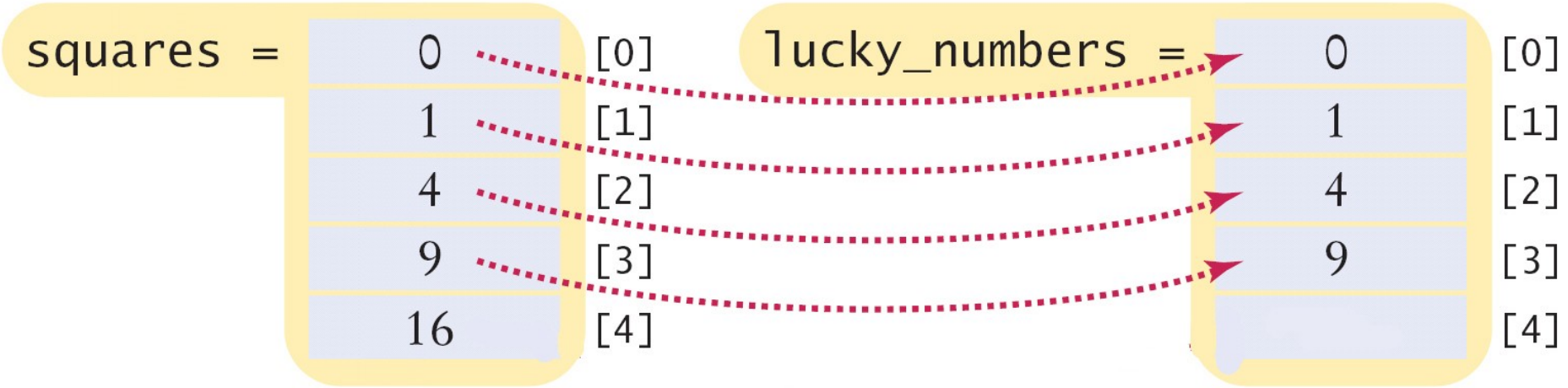
when `i` is 3



Common Algorithms – Copying

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

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

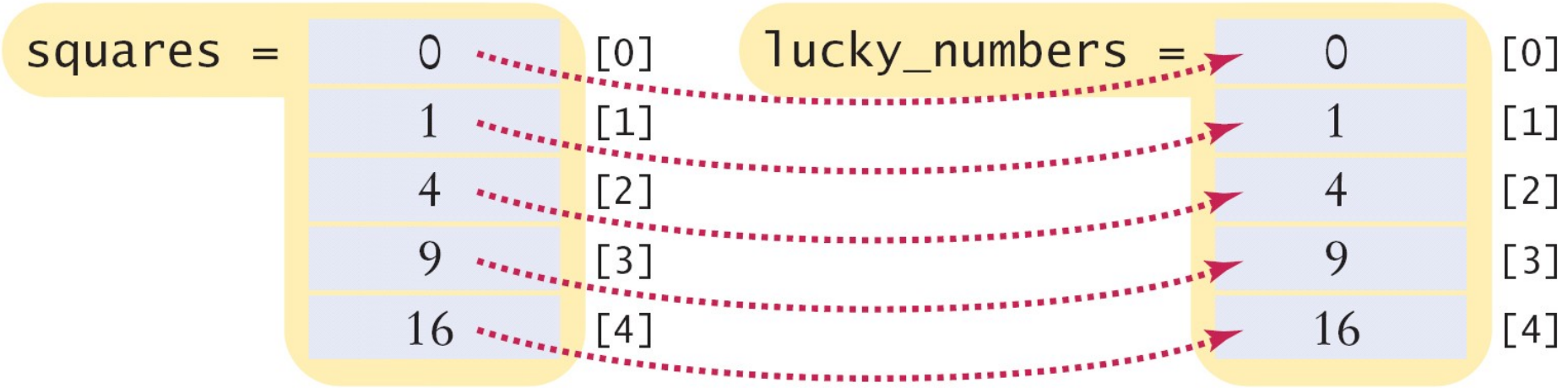
when `i` is `4`



Common Algorithms – Copying

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

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

when `i` is `4`



Common Algorithms – Sum and Average Value

You have already seen the algorithm
for computing the sum and average of 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 of values; i++) {
   total = total + values[i];
}
```

The average is just arithmetic:

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

Common Algorithms – Who Is the Tallest?

If everyone's height is stored in an array,
determining the largest value
(what's the tallest person's height?)
is just another algorithm...

Common Algorithms – Maximum and Minimum

To compute the largest value in an array, keep a variable that stores the largest element that you have encountered, and update it when you find a larger one.

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

Common Algorithms – Maximum and Minimum

To compute the largest value in an array, keep a variable that stores the largest element that you have encountered, and update it when you find a larger one.

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

Note that the loop starts at `1` because we initialize `largest` with `data[0]`.

Common Algorithms – Who Is the Shortest?

Who's the shortest in the line?

(What'is the shortest person's height?)

Common Algorithms – Maximum and Minimum

For the minimum, we just reverse the comparison.

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

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

Common Algorithms – Element Separators

When you display the elements of an array, you usually want to separate them, often with commas or vertical lines, like this:

```
1 | 4 | 9 | 16 | 25
```

Note that there is one fewer separator than there are numbers.

To print five elements,
you need *four* separators.

Common Algorithms – Element Separators

Print the separator before each element *except the initial one* (with index 0):

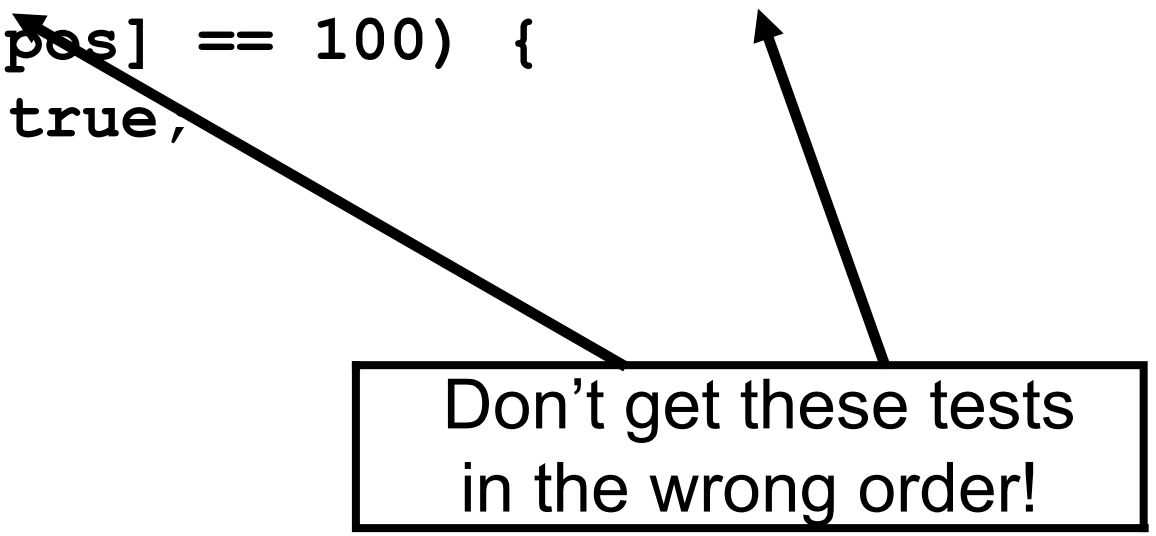
```
1 | 4 | 9 | 16 | 25
```

```
for (int i = 0; i < size of values; i++) {
   if (i > 0) {
      printf(" | ");
   }
   printf("%d", values[i]);
}
```

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 of values && !found) {
   if (values[pos] == 100) {
      found = true;
   } else {
      pos++;
   }
}
```

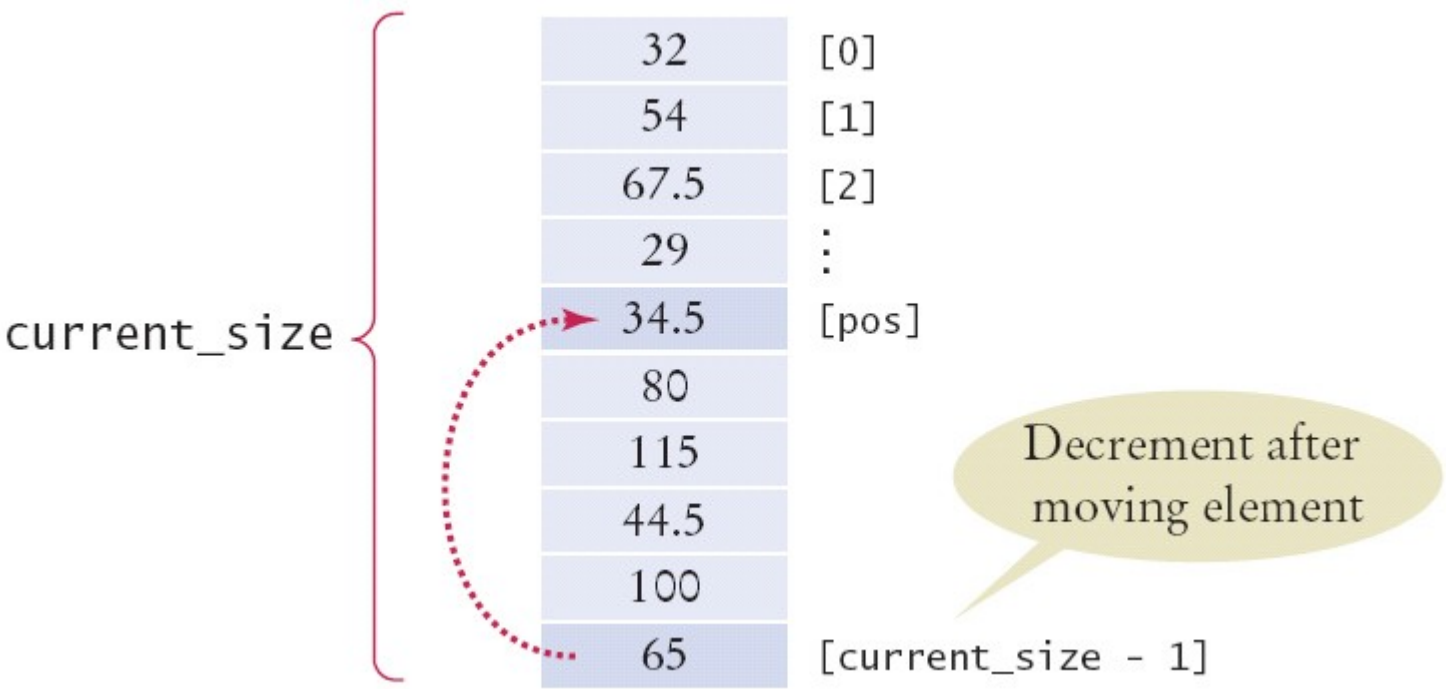


Common Algorithms – Removing an Element, Unordered

Suppose you want to remove the element at index `i`.

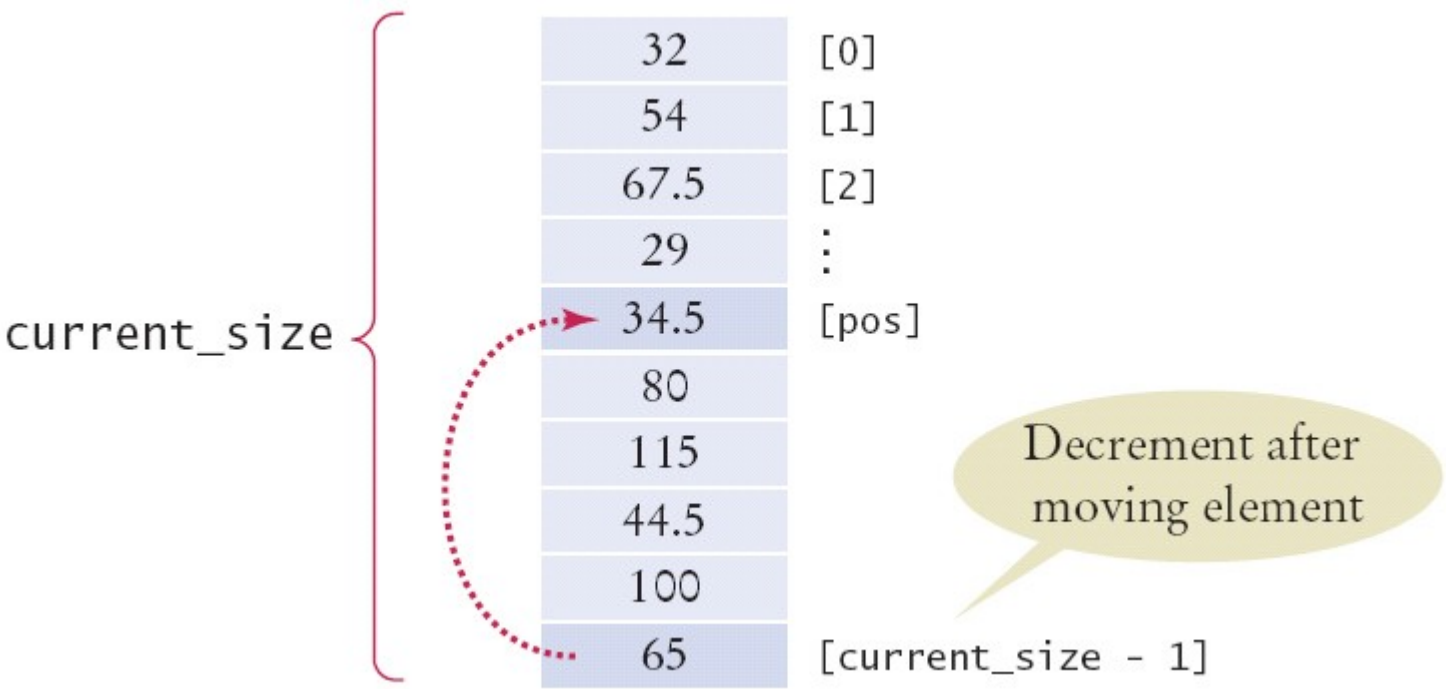
If the elements in the array are not in any particular order, that task is easy to accomplish.

Simply overwrite the element to be removed with the *last* element of the array, then remove the value that was copied by shrinking the size of the array.



Common Algorithms – Removing an Element, Unordered

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

current_size--;
```



Common Algorithms – Removing an Element, Ordered

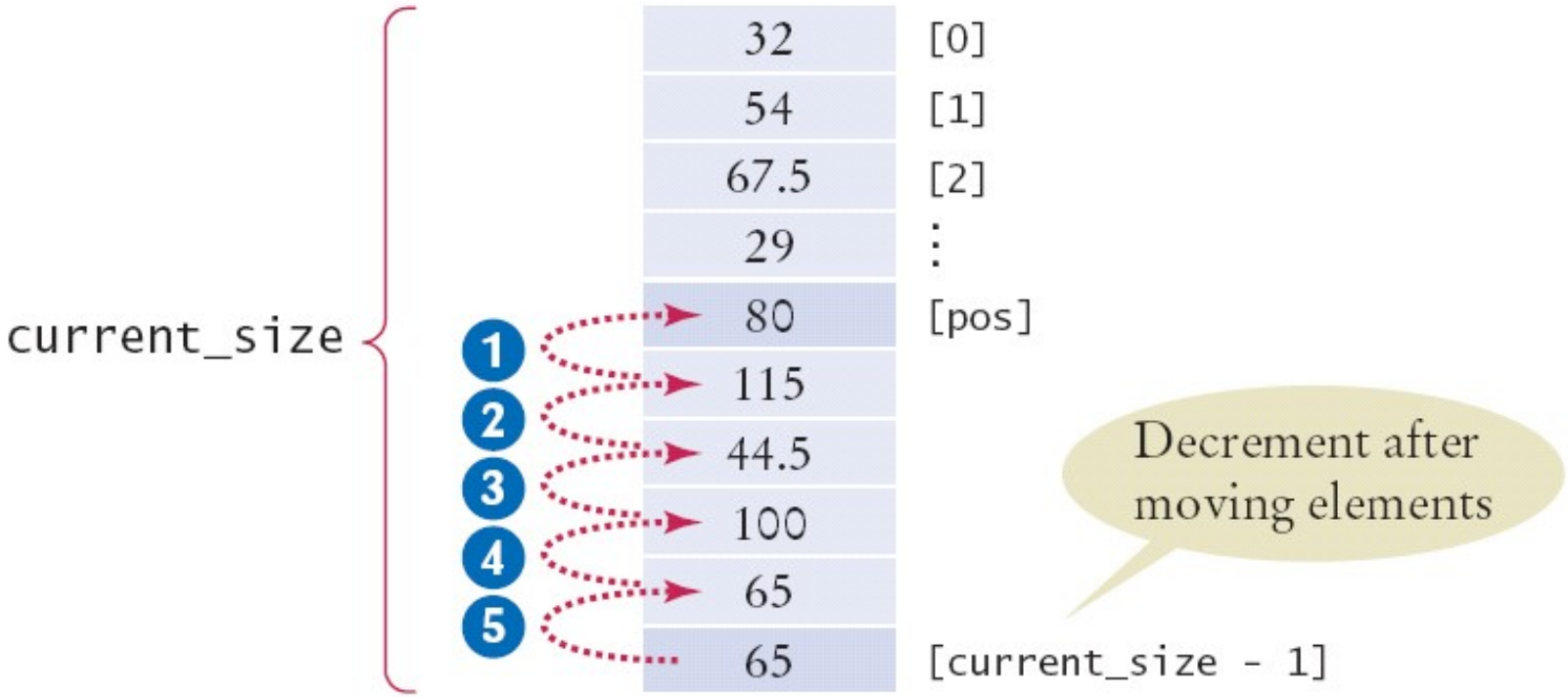
The situation is more complex
if the order of the elements matters.

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

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

Common Algorithms – Removing an Element, Ordered

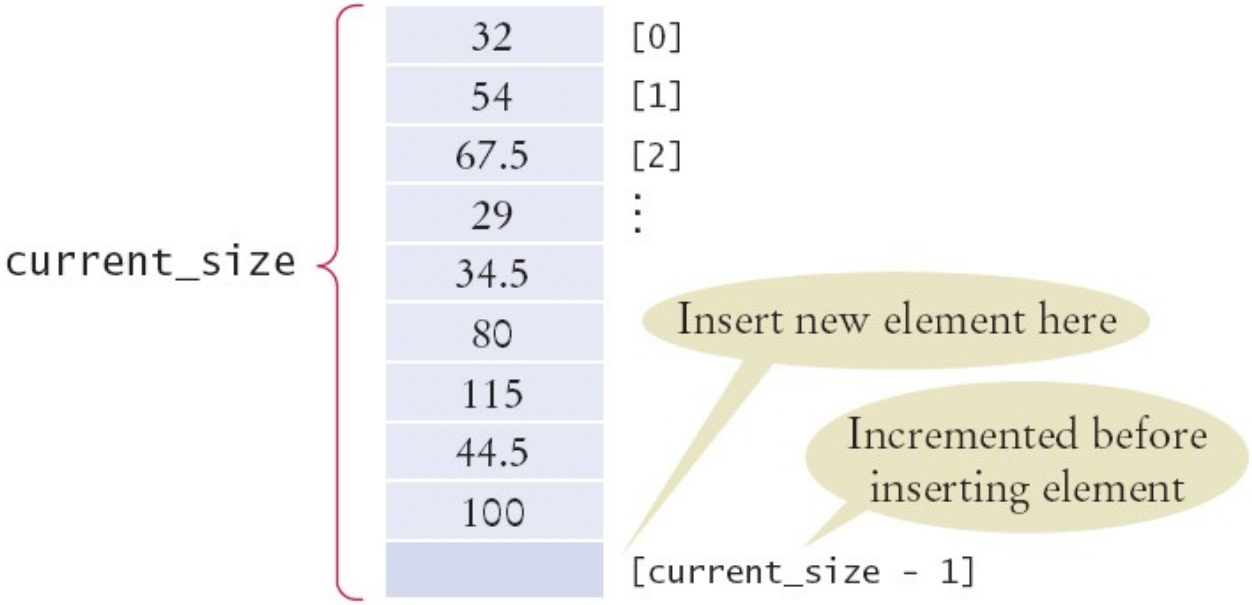
```
for (int i = pos + 1; i < current_size; i++) {
   values[i - 1] = values[i];
}
current_size--;
```



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;
}
```

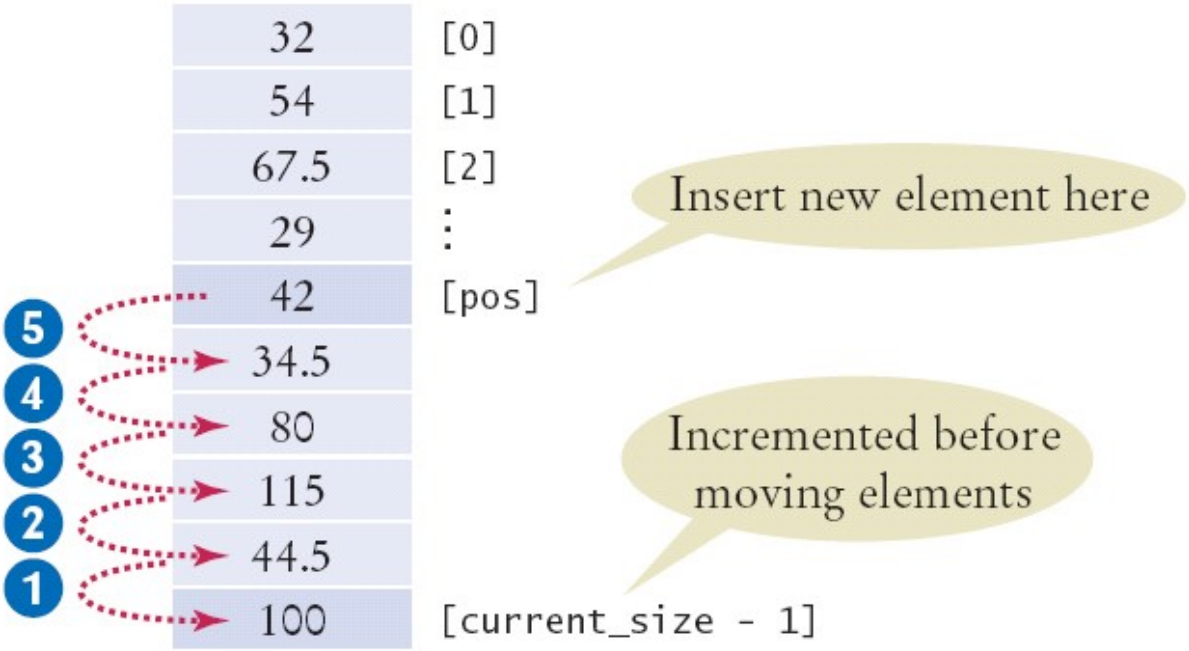


Common Algorithms – Inserting an Element Ordered

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 array must be moved "up".

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



Common Algorithms – Inserting an Element Ordered

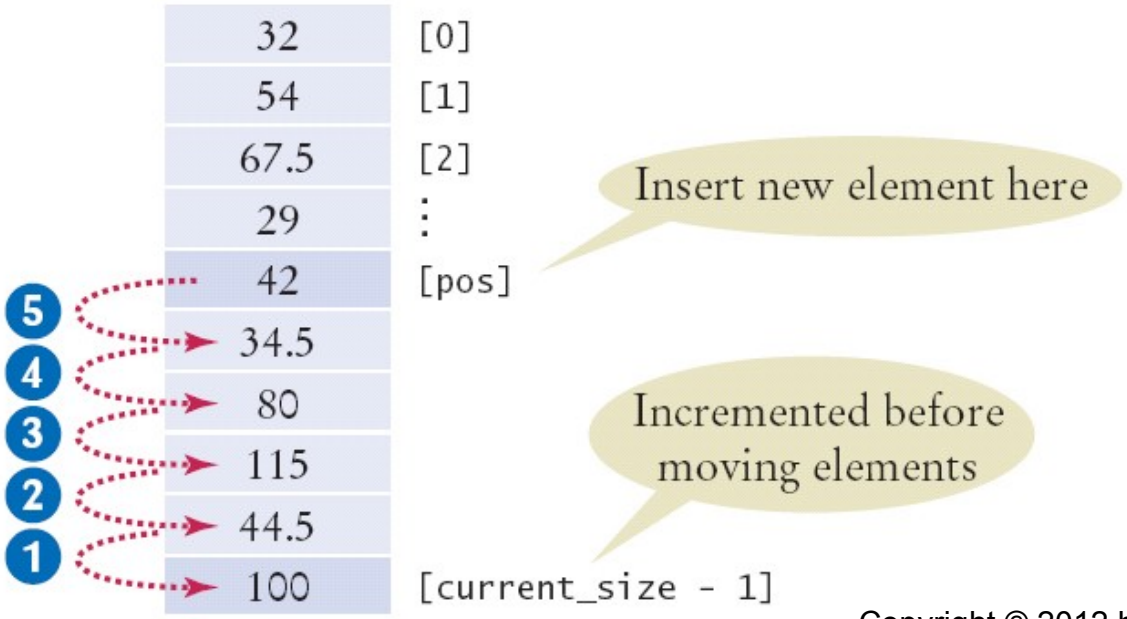
First, you must make the array one larger by incrementing **`current_size`**.

Next, move all elements above the insertion location to a higher index.

Finally, insert the new element in the place you made for it.

Common Algorithms – Inserting an Element Ordered

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



Common Algorithms – Swapping Elements

Swapping two elements in an array
is an important part of sorting an array.

To do a swap of two things,
you need *three* things.

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?

Common Algorithms – Swapping Elements

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

STEP One

save the value at `i`

STEP Two

replace the value at `i`

STEP Three

now you can change the value at `j` because you saved from `i`

Common Algorithms – Reading Input

If the know how many input values the user will supply, you can store them directly into the array:

```
double values[NUMBER_OF_INPUTS];
for (i = 0; i < NUMBER_OF_INPUTS; i++) {
   printf("%lf", &values[i]);
}
```

Common Algorithms – Reading Input

When there will be an arbitrary number of inputs, things get more complicated.

Add values to the end of the array until all inputs have been made.

Again, the companion variable will have the number of inputs.

```
double values[CAPACITY];
int current_size = 0;
double input;
scanf("%lf", &input);
while (input > 0) {
   if (current_size < CAPACITY) {
      values[current_size] = input;
      current_size++;
   }
   scanf("%lf", &input);
}
```

Common Algorithms – Reading Input

Unfortunately it's even more complicated:

Once the array is full, we allow the user to keep entering!

Because we can't change the size
of an array after it has been created,
we'll just have to give up for now.

Common Algorithms

Now back to where we started:

How do we determine the largest in a set of data?

Common Algorithms – Maximum

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   const int CAPACITY = 1000;
   double values[CAPACITY];
   int current_size = 0;
```

Common Algorithms – Maximum

```
printf("Please enter values, 0 to quit:\n");
double input;
scanf("%lf", &input);
while (input > 0) {
   if (current_size < CAPACITY) {
      values[current_size] = input;
      current_size++;
   }
   scanf("%lf", &input);
}
```

Common Algorithms – Maximum

```
double largest = values[0];
for (int i = 1; i < current_size; i++) {
   if (values[i] > largest) {
      largest = values[i];
   }
}
```

Common Algorithms – Maximum

```
   for (int i = 0; i < current_size; i++) {
      printf(" %f ", values[i]);
      if (values[i] == largest) {
         printf(" (largest value) ");
      }
      printf("\n");
   }

   return EXIT_SUCCESS;
}
```

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.

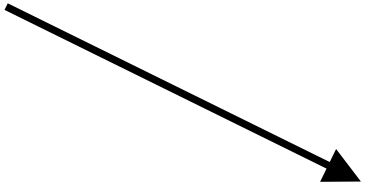
Arrays as Parameters in Functions

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;
}
```

Arrays as Parameters in Functions

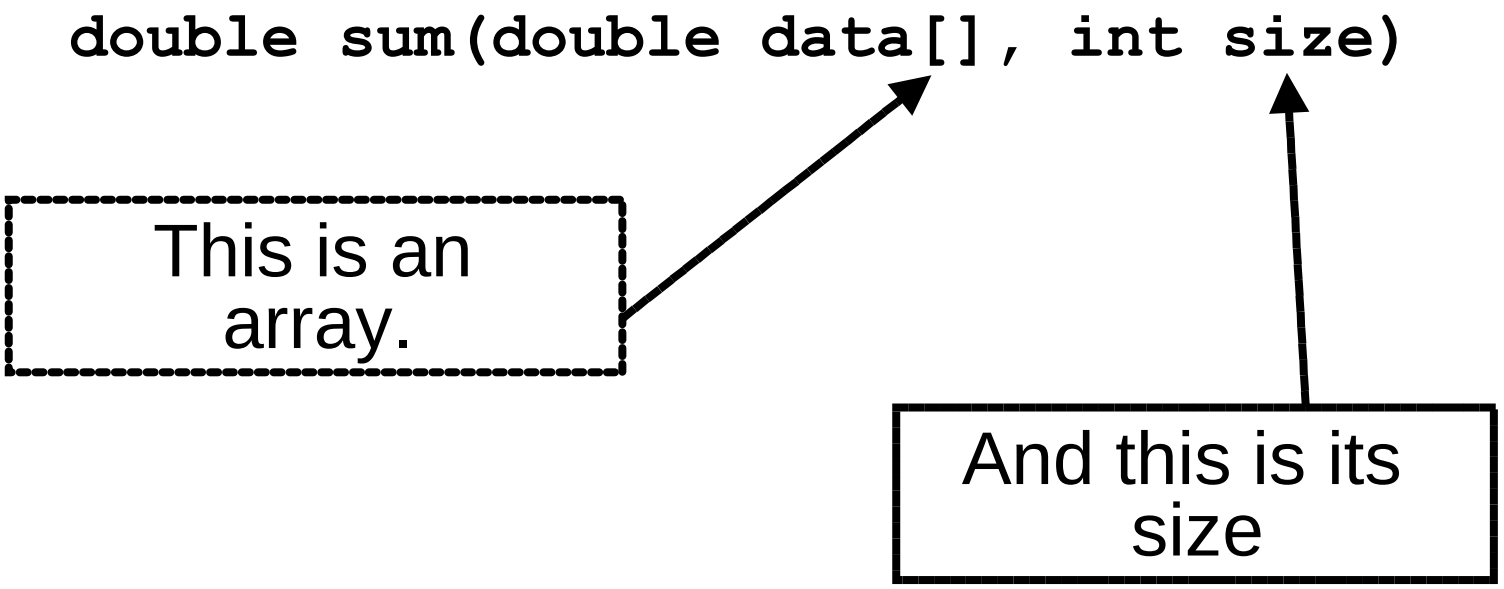
empty pair of square brackets



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

Arrays as Parameters in Functions

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



Arrays as Parameters in Functions

When you call the function,
supply both the name of the array and the size:

```
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);
```

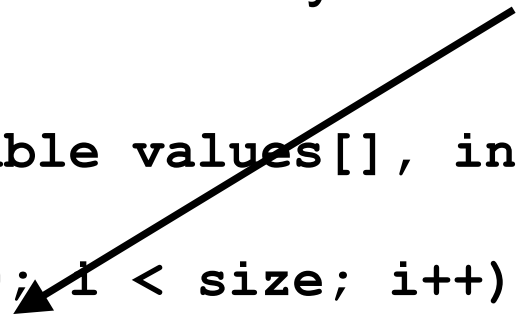
You can also pass a smaller size to the function:

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

This will sum over only the first five **doubles** in the array.

Arrays as Parameters in Functions

When you pass an array into a function,
the contents of the array can ***always*** be changed:



```
void multiply(double values[], int size, double factor)
{
   for (int i = 0; i < size; i++) {
      values[i] = values[i] * factor;
   }
}
```

Arrays as Parameters in Functions

You can pass an array into a function

but

you cannot return an array.

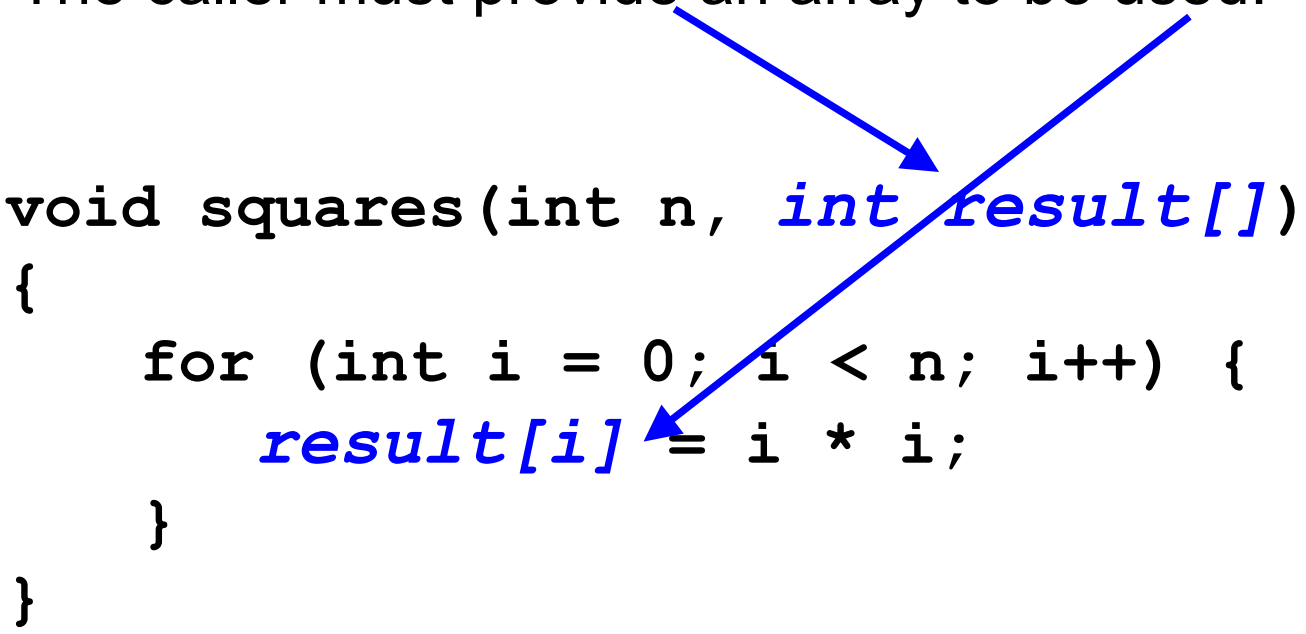
Arrays as Parameters in Functions

If you cannot return an array, how can the caller get the data?

```
??? squares(int n)
{
   int result[]
   for (int i = 0; i < n; i++) {
      result[i] = i * i;
   }
   return result; // ERROR
}
```

Arrays as Parameters in Functions

The caller must provide an array to be used:

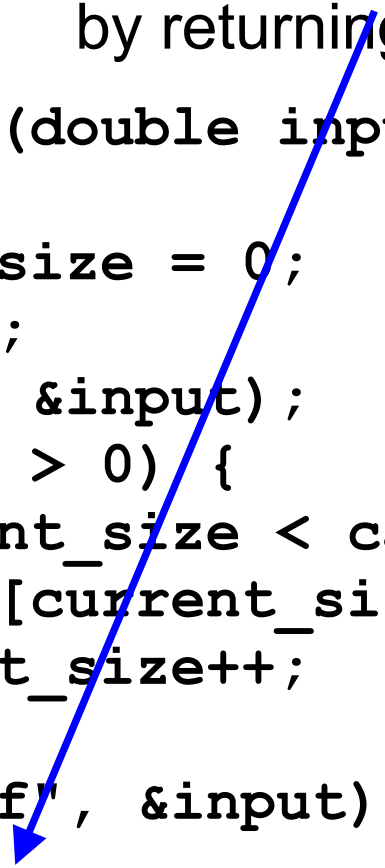


```
void squares(int n, int result[])
{
   for (int i = 0; i < n; i++) {
      result[i] = i * i;
   }
}
```

Arrays as Parameters in Functions

A function can change the size of an array.

It should let the caller know of any change by returning the new size.



```
int read_inputs(double inputs[], int capacity)
{
   int current_size = 0;
   double input;
   scanf("%lf", &input);
   while (input > 0) {
      if (current_size < capacity) {
         inputs[current_size] = input;
         current_size++;
      }
      scanf("%lf", &input);
   }
   return current_size;
}
```

Arrays as Parameters in Functions

Here is a call to the function:

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

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

Arrays as Parameters in Functions

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.

Arrays as Parameters in Functions

```
#include <stdio.h>
#include <stdlib.h>



/**
 * Reads a sequence of floating-point numbers.
 *
 * @param inputs an array containing the numbers
 * @param capacity the capacity of that array
 * @return the number of inputs stored in the array
 */
int read_inputs(double inputs[], int capacity)
{
```

Arrays as Parameters in Functions

```
   int current_size = 0;
   printf("Please enter values, 0 to quit:\n");
   bool more = true;
   while (more) {
      double input;
       scanf("%lf", &input);
       if (input <= 0) {
          more = false;
       } else if (current_size < capacity) {
          inputs[current_size] = input;
          current_size++;
       }
   }
   return current_size;
}
```

Arrays as Parameters in Functions

```
/**
 * Multiplies all elements of an array by a factor.
 *
 * @param values a partially filled array
 * @param size the number of elements in values
 * @param factor the value with which each element is
  multiplied
 */
void multiply(double values[], int size,
              double factor)
{
   for (int i = 0; i < size; i++) {
      values[i] = values[i] * factor;
   }
}
```

Arrays as Parameters in Functions

```
/**
   Prints the elements of an array, separated by
   commas.
   @param values a partially filled array
   @param size the number of elements in values
*/
void print(double values[], int size)
{
   for (int i = 0; i < size; i++) {
      if (i > 0) {
         printf(", ");
      }
      printf("%f", values[i]);
   }
   printf("\n");
}
```

Arrays as Parameters in Functions

```
int main()
{
   const int CAPACITY = 1000;
   double values[CAPACITY];
   int size = read_inputs(values, CAPACITY);
   multiply(values, size, 2);
   print(values, size);

   return EXIT_SUCCESS;
}
```

Problem Solving: Adapting Algorithms

You can try to use algorithms you already know
to produce a new algorithm that will solve this problem.

(Then you'll have yet another algorithm.)

Problem Solving: Adapting Algorithms

Consider this problem:

Compute the final quiz score from a set of quiz scores,

but be nice:

drop the lowest score.

Problem Solving: Adapting Algorithms

What do I know how to do?

Problem Solving: Adapting Algorithms

Calculate the sum:

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

Problem Solving: Adapting Algorithms

Find the minimum:

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

Problem Solving: Adapting Algorithms

Remove an element:

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

Problem Solving: Adapting Algorithms

Here is the algorithm:

1. *Find the minimum*
2. *Remove it from the array*
3. *Calculate the sum*
   (will be without the lowest score)
4. *Calculate the final score*

Problem Solving: Adapting Algorithms

WAIT!

Problem Solving: Adapting Algorithms

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

This algorithm removes by knowing
the position
of the element to remove…

…but…

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

That's not the *position* of the smallest –
it IS the smallest.

Problem Solving: Adapting Algorithms

Here's another algorithm I know that *does* find the position:

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

Problem Solving: Adapting Algorithms

Here is the algorithm:

1. *Find the minimum*
2. *Find the position of the minimum*
   → **the one I just searched for!!!**
3. *Remove it from the array*
4. *Calculate the sum*
   (will be without the lowest score)
5. *Calculate the final score*

Problem Solving: Adapting Algorithms

But I'm repeating myself.

I searched
for the minimum
and then
I searched
for the position…
…of the minimum!

Problem Solving: Adapting Algorithms

I wonder if I can *adapt* the algorithm that finds the minimum so that it finds the position of the minimum?

Problem Solving: Adapting Algorithms

Start with this:

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

Problem Solving: Adapting Algorithms

What is it about the minimum value
and where the minimum value is?

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

Problem Solving: Adapting Algorithms

What is it about the minimum value
and where the minimum value is?

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

Problem Solving: Adapting Algorithms

What is it about the minimum value
and where the minimum value is?

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

Problem Solving: Adapting Algorithms

What is it about the minimum value
and where the minimum value is?

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

Problem Solving: Adapting Algorithms

What is it about the minimum value
and where the minimum value is?

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

Problem Solving: Adapting Algorithms

What is it about the minimum value
and where the minimum value is?

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

Problem Solving: Adapting Algorithms

There it is!

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

Problem Solving: Adapting Algorithms

Finally:

1. *Find the* **position** *of the minimum*
2. *Remove it from the array*
3. *Calculate the sum*
   (will be without the lowest score)
4. *Calculate the final score*

Discovering Algorithms by Manipulating Physical Objects

There is a technique that you can use called:

MANIPULATING PHYSICAL OBJECTS

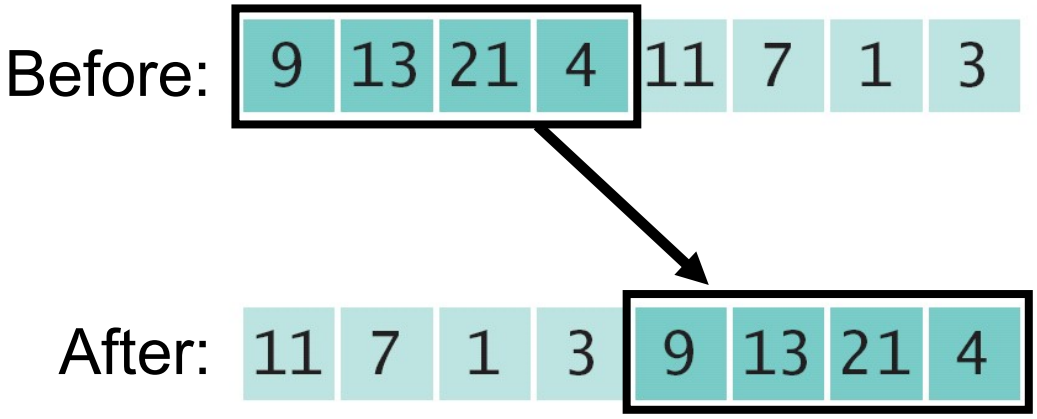
better know as:

playing around with things.

Discovering Algorithms by Manipulating Physical Objects

Here is a problem:

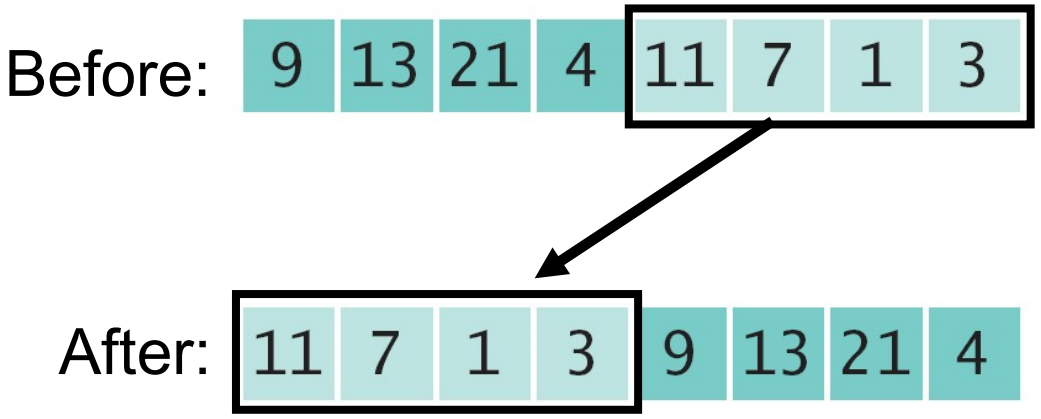
You are given an array whose size is an even number.
You are to switch the first and the second half.



Discovering Algorithms by Manipulating Physical Objects

Here is a problem:

You are given an array whose size is an even number.
You are to switch the first and the second half.



Discovering Algorithms by Manipulating Physical Objects

To learn this *Manipulating Physical Objects* technique,
let's play with some coins
and review some algorithms you already know.

OK, let's *manipulate* some coins.

Go get eight coins.

Discovering Algorithms by Manipulating Physical Objects

Discovering Algorithms by Manipulating Physical Objects

What algorithms do you know
that allow you to rearrange a set of coins?

Discovering Algorithms by Manipulating Physical Objects

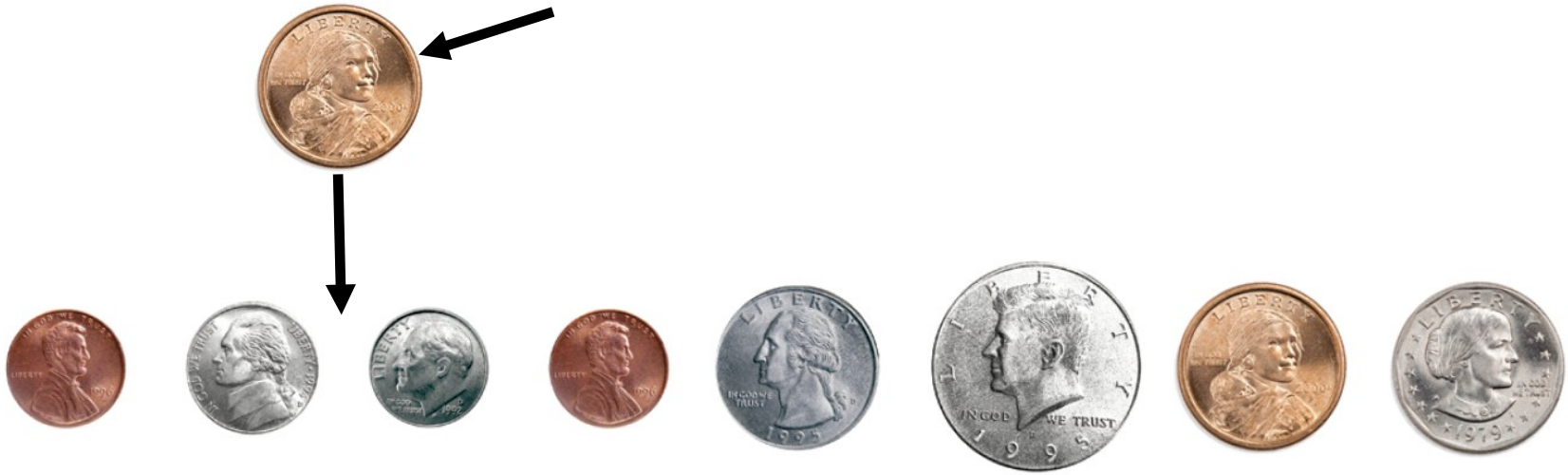
You know how to remove a coin.

Discovering Algorithms by Manipulating Physical Objects

You know how to remove a coin.

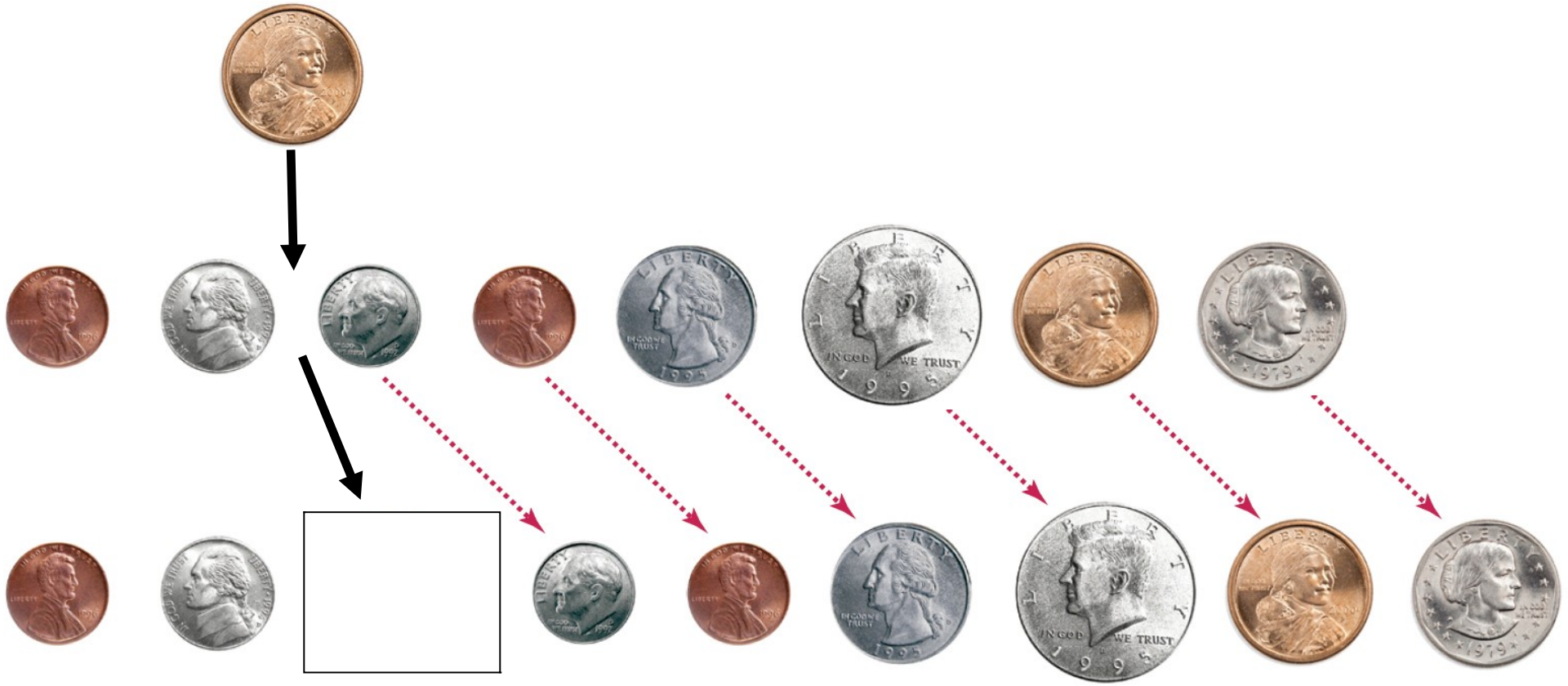
Discovering Algorithms by Manipulating Physical Objects

You know how to insert a coin at a specific position.



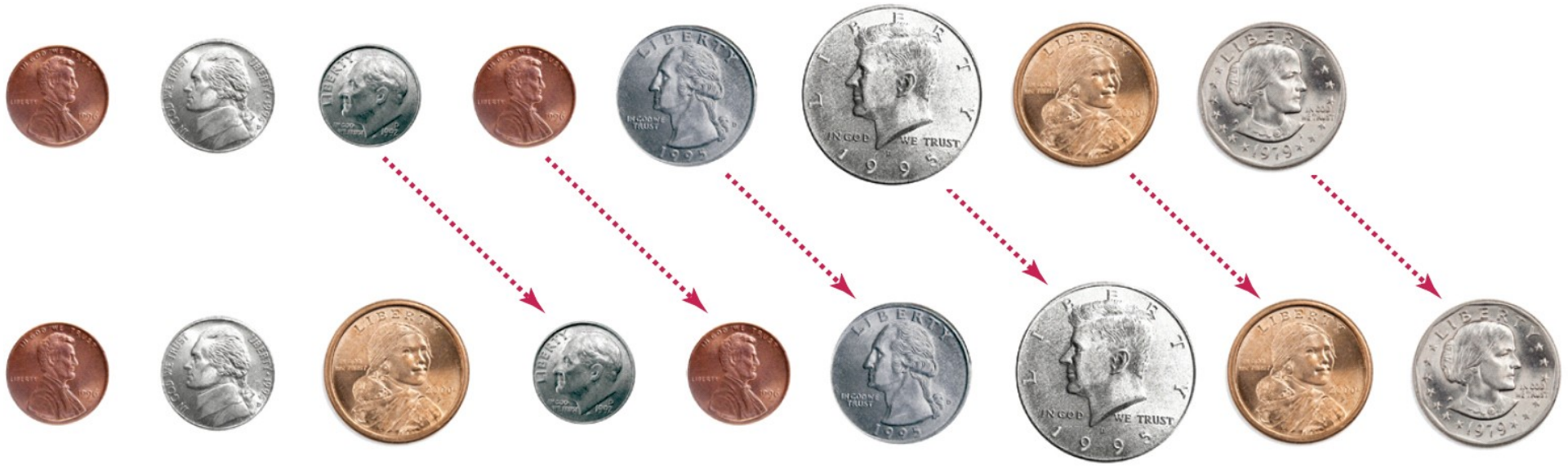
Discovering Algorithms by Manipulating Physical Objects

You know how to insert a coin at a specific position.



Discovering Algorithms by Manipulating Physical Objects

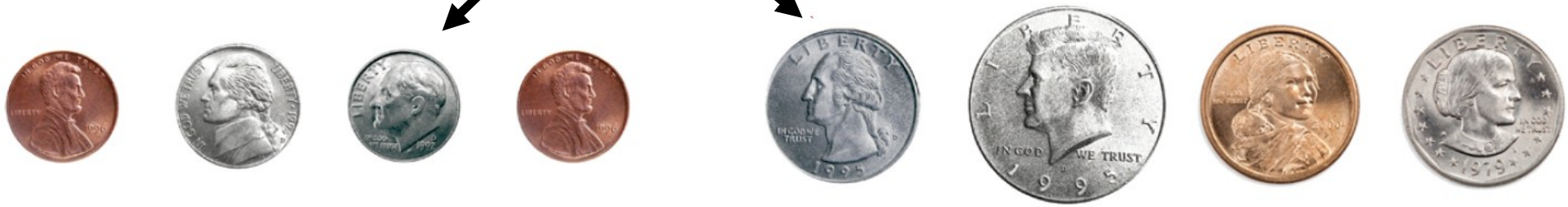
You know how to insert a coin at a specific position.



Discovering Algorithms by Manipulating Physical Objects

And you know how to swap two elements.

You two!



Swap places!

Discovering Algorithms by Manipulating Physical Objects

And you know how to swap two elements.

Discovering Algorithms by Manipulating Physical Objects

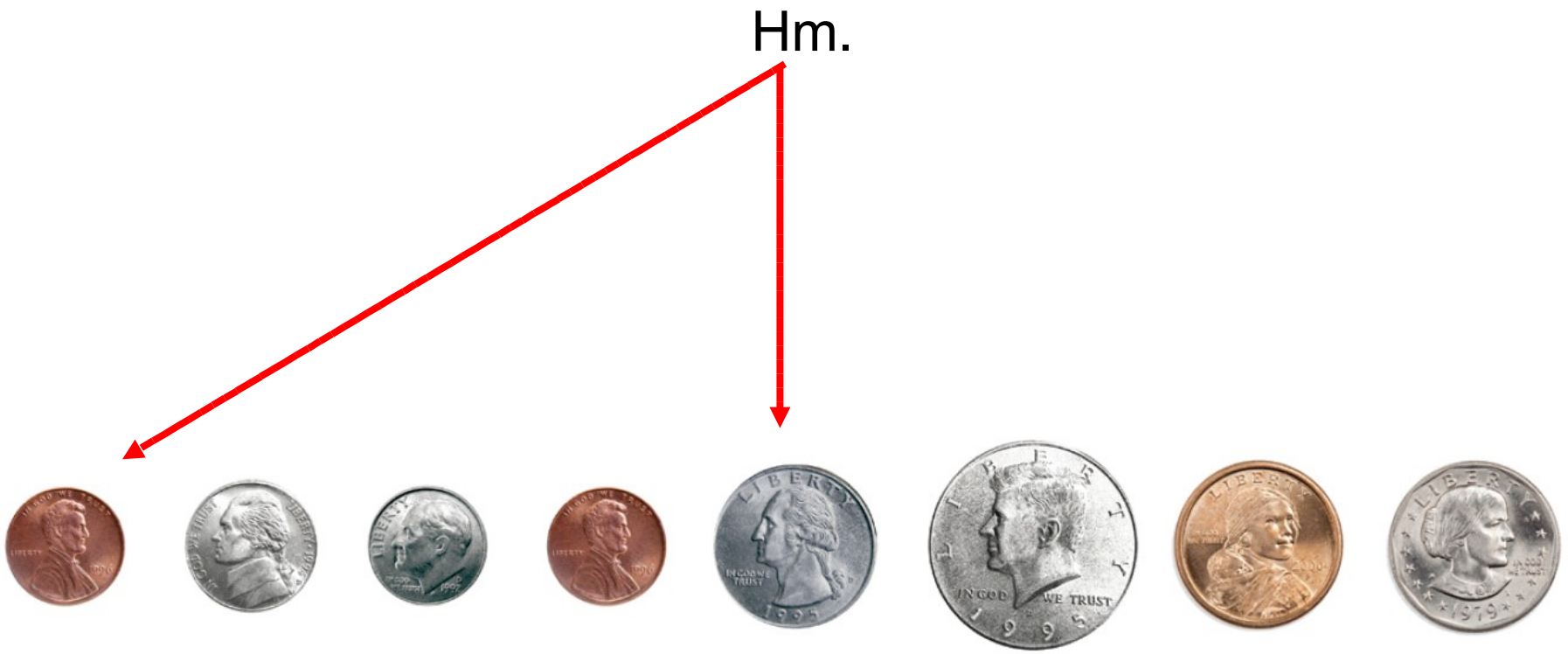
And you know how to swap two elements.

Discovering Algorithms by Manipulating Physical Objects

Discovering Algorithms by Manipulating Physical Objects

Discovering Algorithms by Manipulating Physical Objects

Discovering Algorithms by Manipulating Physical Objects



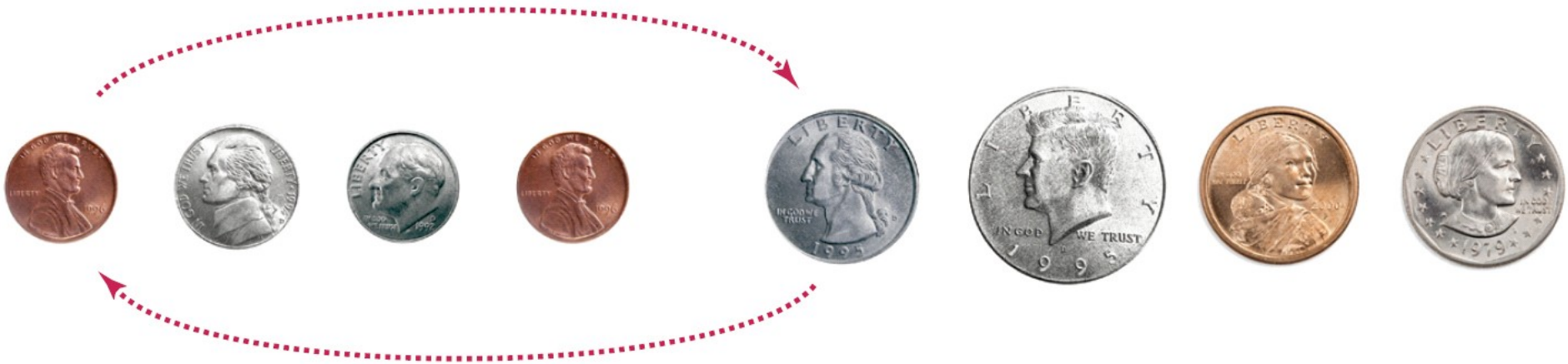
Discovering Algorithms by Manipulating Physical Objects

Discovering Algorithms by Manipulating Physical Objects

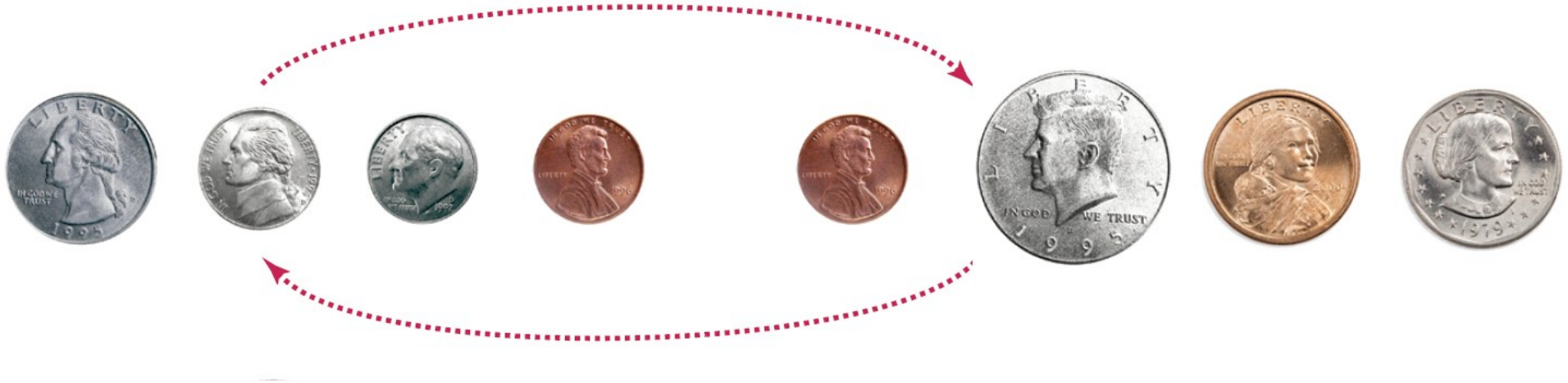
Then hm again.

Discovering Algorithms by Manipulating Physical Objects

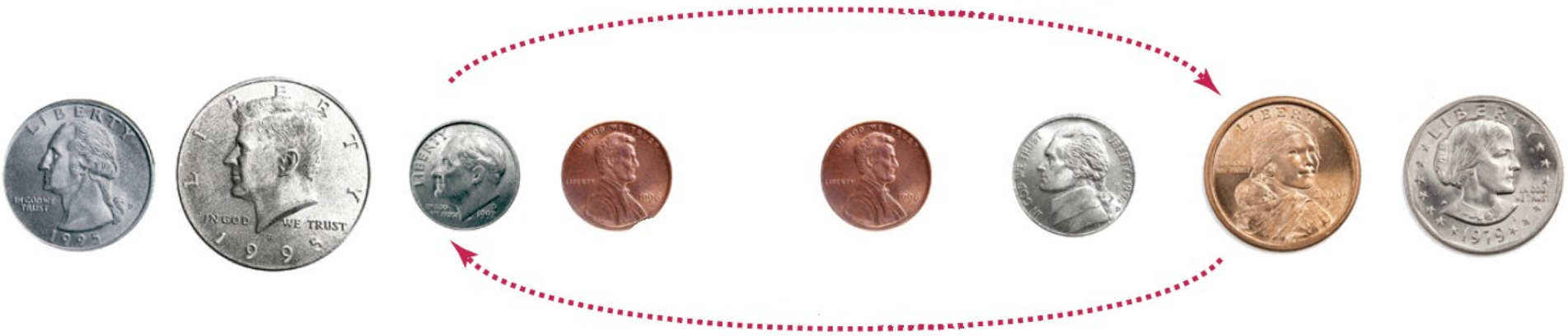
Discovering Algorithms by Manipulating Physical Objects



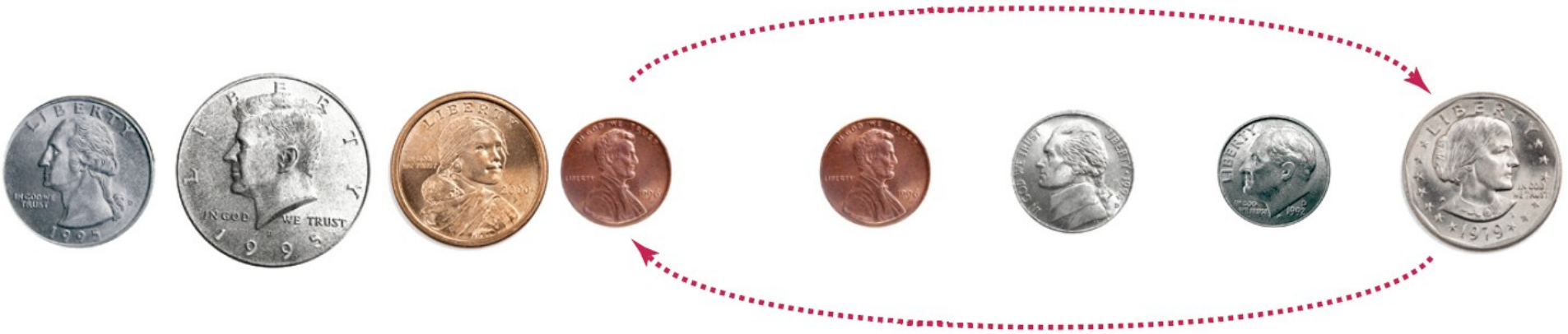
Discovering Algorithms by Manipulating Physical Objects



Discovering Algorithms by Manipulating Physical Objects

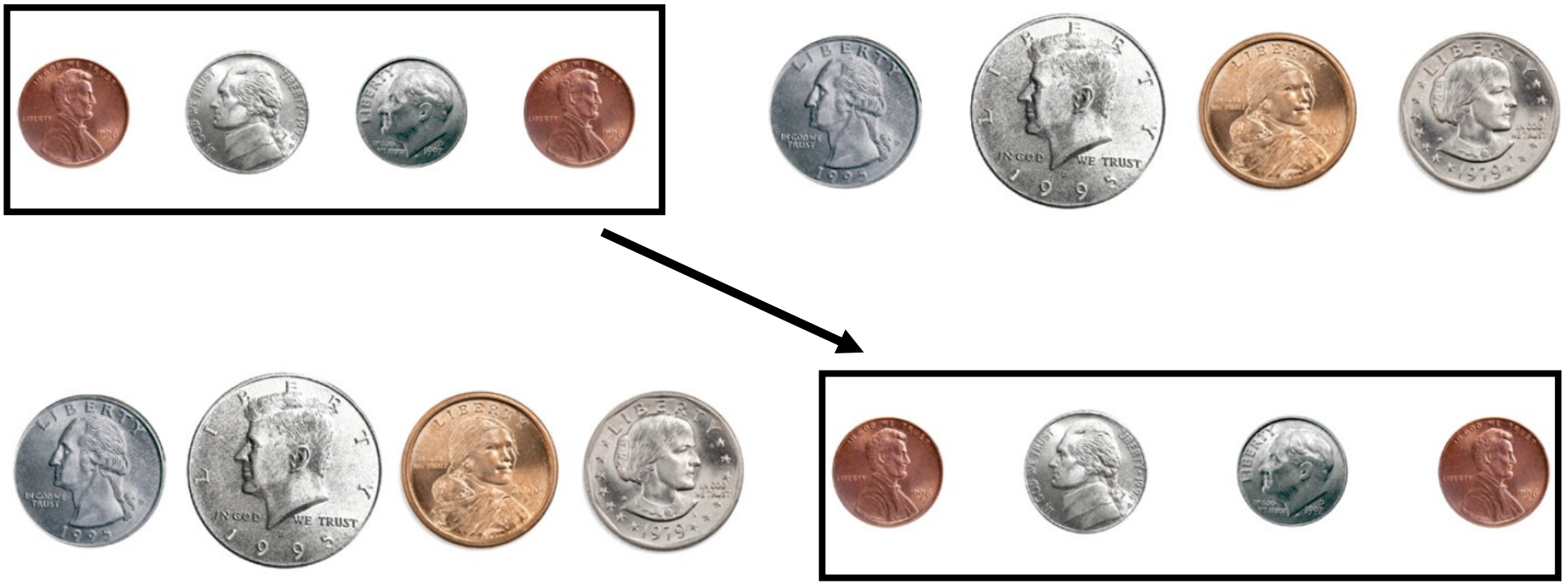


Discovering Algorithms by Manipulating Physical Objects

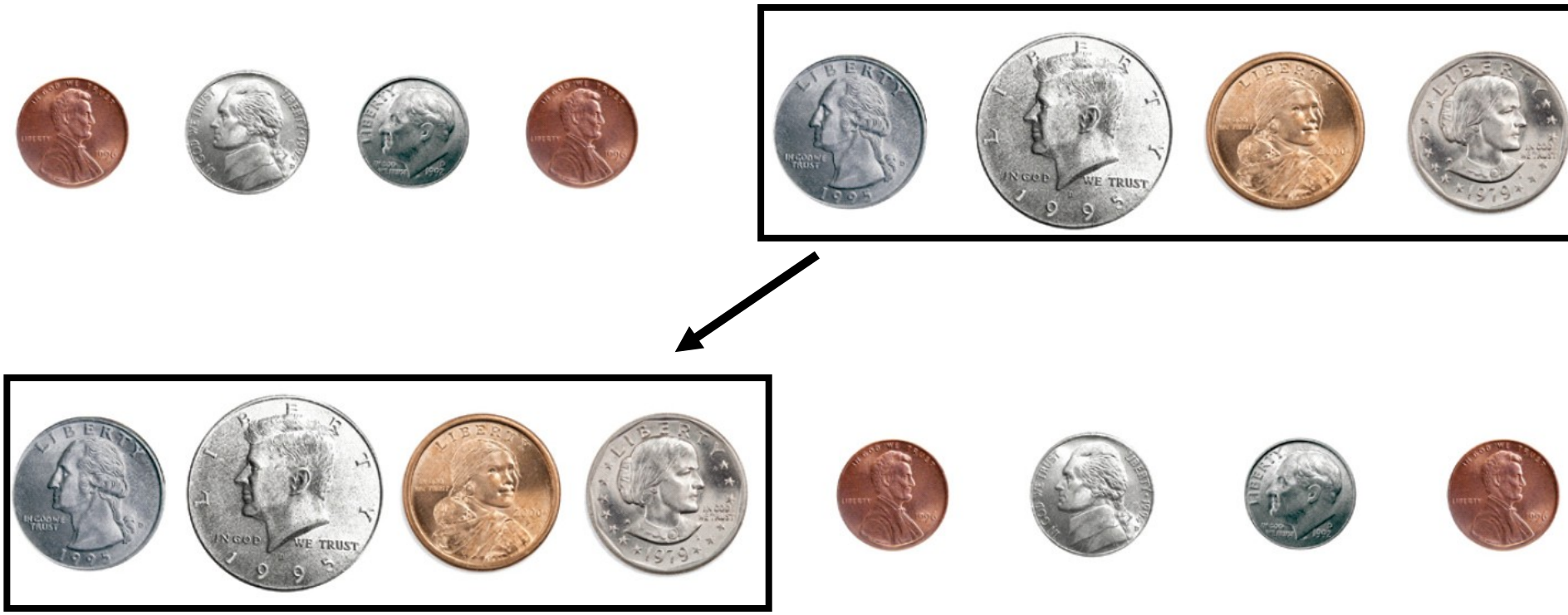


Discovering Algorithms by Manipulating Physical Objects

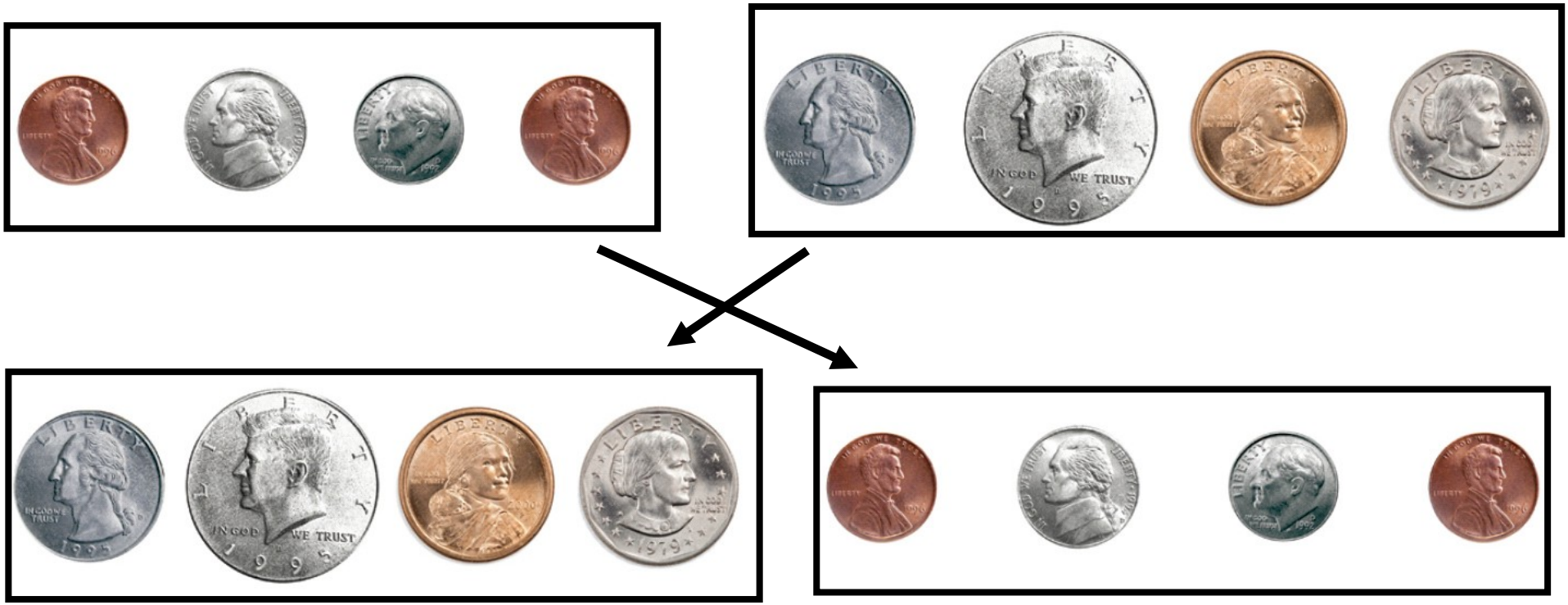
Discovering Algorithms by Manipulating Physical Objects



Discovering Algorithms by Manipulating Physical Objects



Discovering Algorithms by Manipulating Physical Objects



Discovering Algorithms by Manipulating Physical Objects

```
i =
j =
```

Two indices means we need two variables.

Discovering Algorithms by Manipulating Physical Objects

```
i =
j =
```

Initialization?

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j =
```

OK.

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j =
```

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = ?
```

Where does that index start?

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = ?
```

We swap the leftmost with somewhere in the middle.

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = ?
```

The middle!
That's it – half way into the array.

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
```

Now we will loop…

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
```

…but…
for how long?
until when?

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
```

Let's think about that later.

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
```

For certain we will be swapping the elements at the indices…

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
    swap elements at i and j
```

…and then go on to the next pair of indices to swap…

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
   swap elements at i and j
   i++;
   j++;
```

But when are we finished swapping?

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
    swap elements at i and j
    i++;
    j++;
```

We only process *half* the array

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while (i < size / 2)
   swap elements at i and j
   i++;
   j++;
```

That's the algorithm!

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.

Two-Dimensional Arrays

An arrangement consisting of *tabular data*:

rows and columns of values

is called:

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

Two-Dimensional Arrays

Consider this data from the 2010 Olympic skating competitions:

	Gold	Silver	Bronze
Canada	1	0	1
China	1	1	0
Germany	0	0	1
Korea	1	0	0
Japan	0	1	1
Russia	0	1	1
United States	1	1	0

Defining Two-Dimensional Arrays

C uses an array with *two* subscripts
to store a *two*-dimensional array.

```
const int COUNTRIES = 7;
const int MEDALS = 3;
int counts[COUNTRIES][MEDALS];
```

An array with 7 rows and 3 columns
is suitable for storing our medal count data.

Defining Two-Dimensional Arrays – Unchangeable Size

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

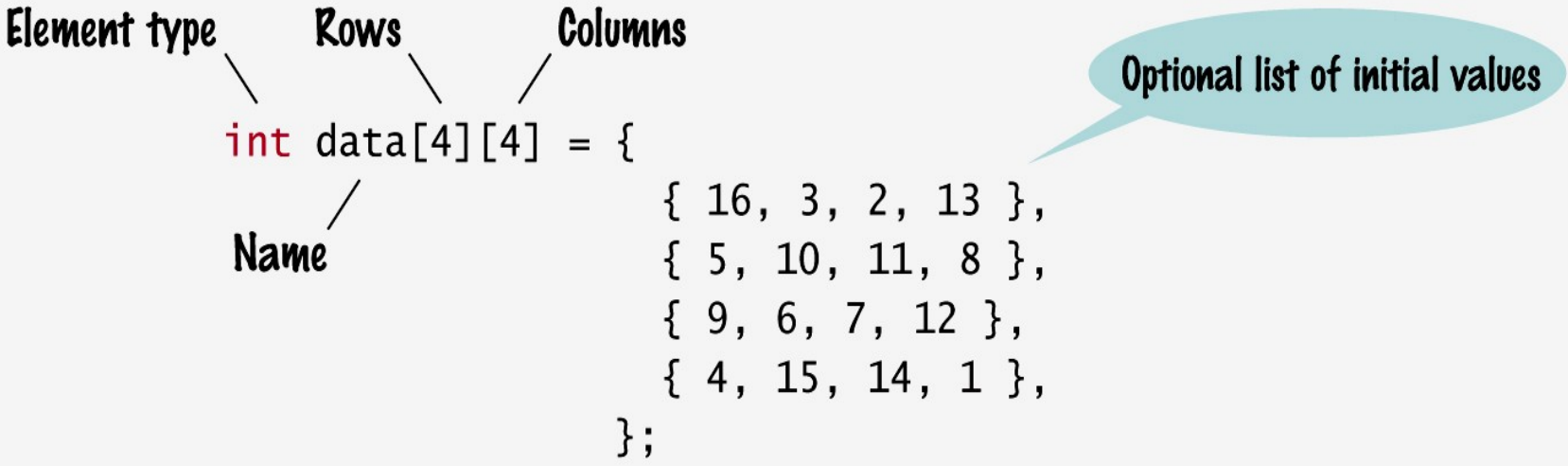
Defining Two-Dimensional Arrays – Initializing

But you can initialize a 2-D array:

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

Defining Two-Dimensional Arrays

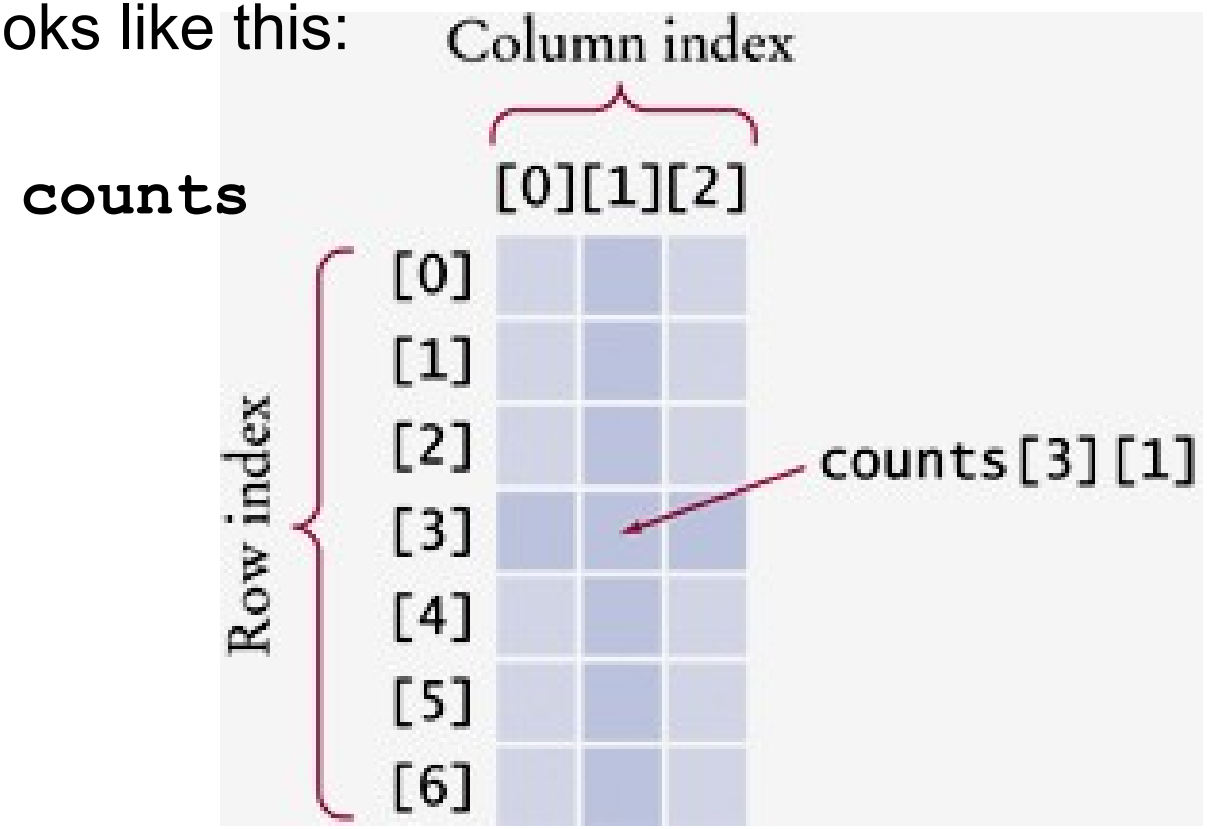
Syntax 6.3 Two-Dimensional Array Definition




```
int data[4][4] = {
      { 16, 3, 2, 13 },
      { 5, 10, 11, 8 },
      { 9, 6, 7, 12 },
      { 4, 15, 14, 1 },
   };
```

Defining Two-Dimensional Arrays – Accessing Elements

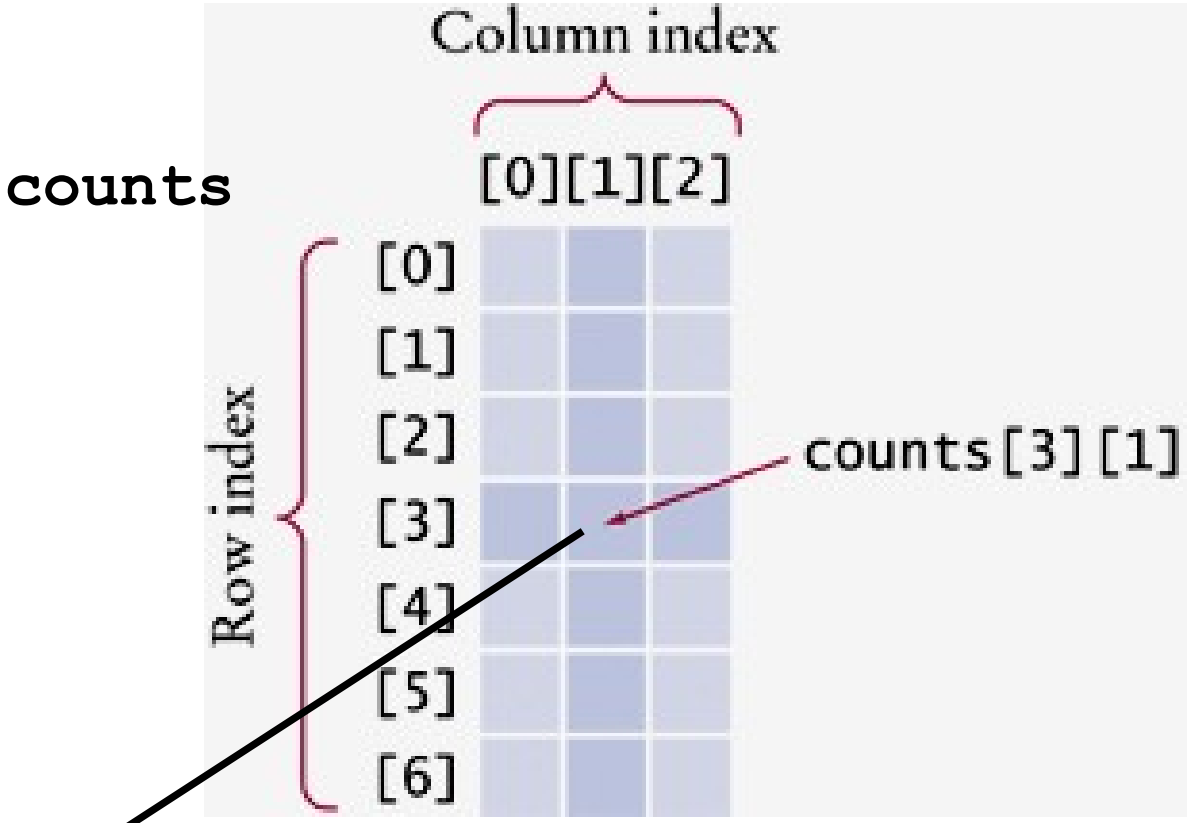
The Olympic array looks like this:



Access to the second element in the fourth row is:

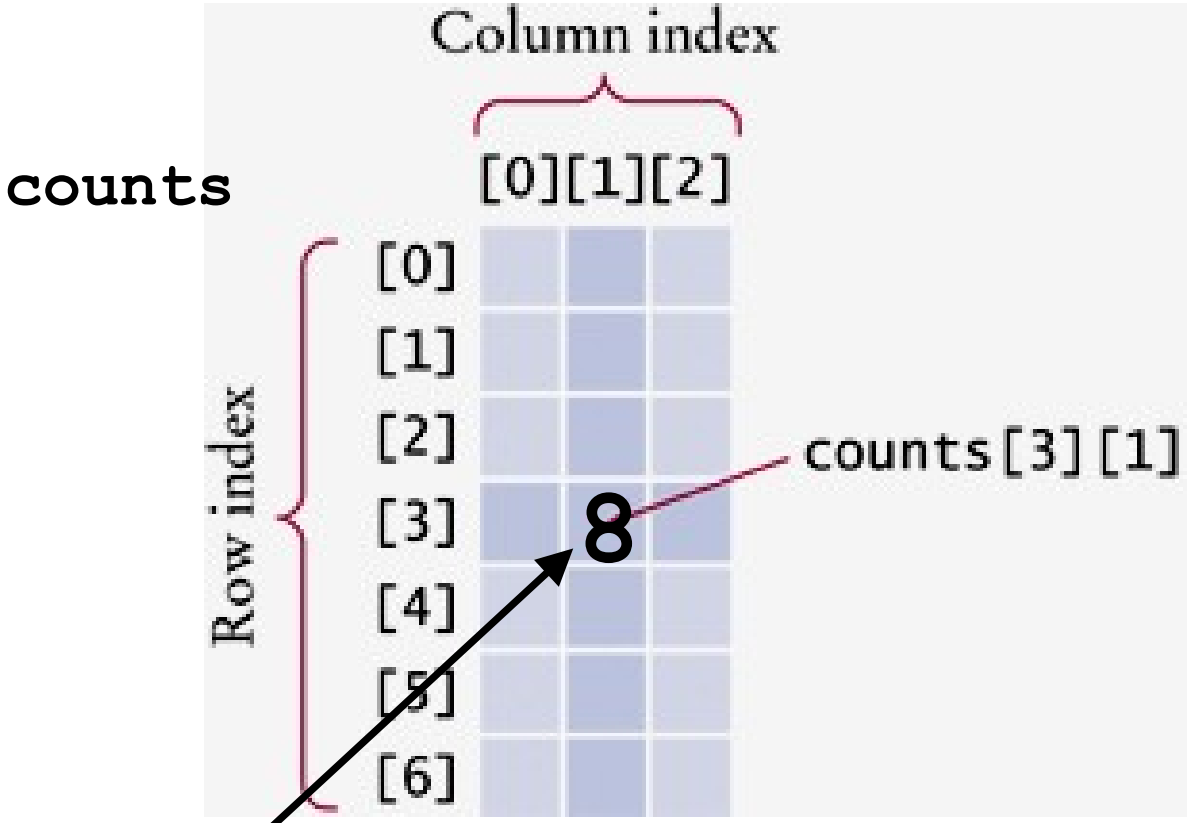
```
counts[3][1]
```

Defining Two-Dimensional Arrays – Accessing Elements




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

Defining Two-Dimensional Arrays – Accessing Elements




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

Two-Dimensional Arrays

```
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
      printf("%8d", counts[i][j]);
   }
   // Start a new line at the end of the row
   printf("\n");
}
```

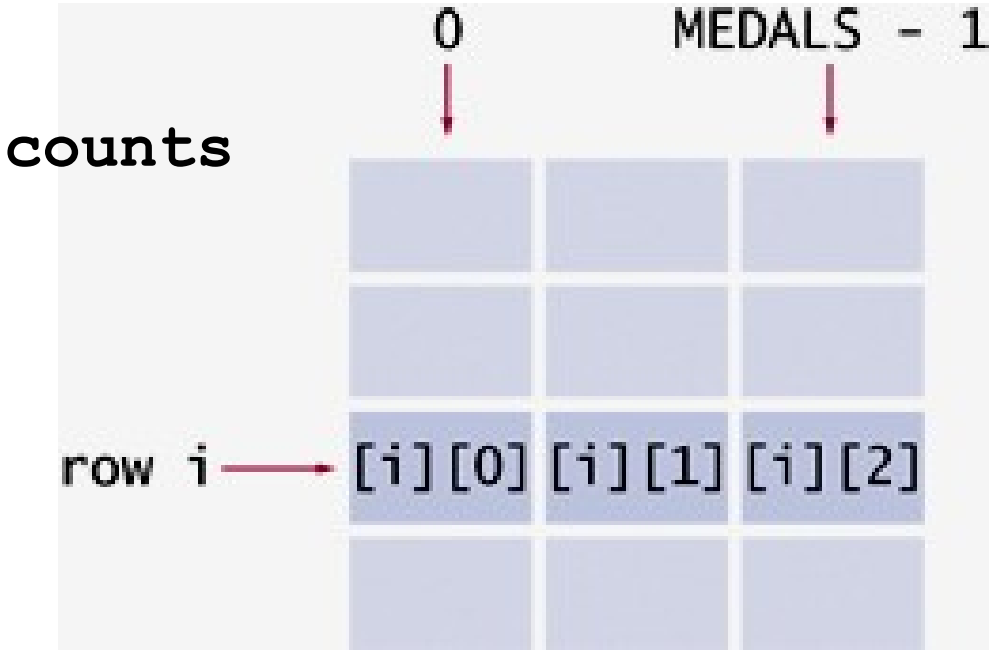
Computing Row and Column Totals

A common task is to compute row or column totals.

In our example,
the row totals give us the total number
of medals won by a particular country.

Computing Row and Column Totals

We must be careful to get the right indices.



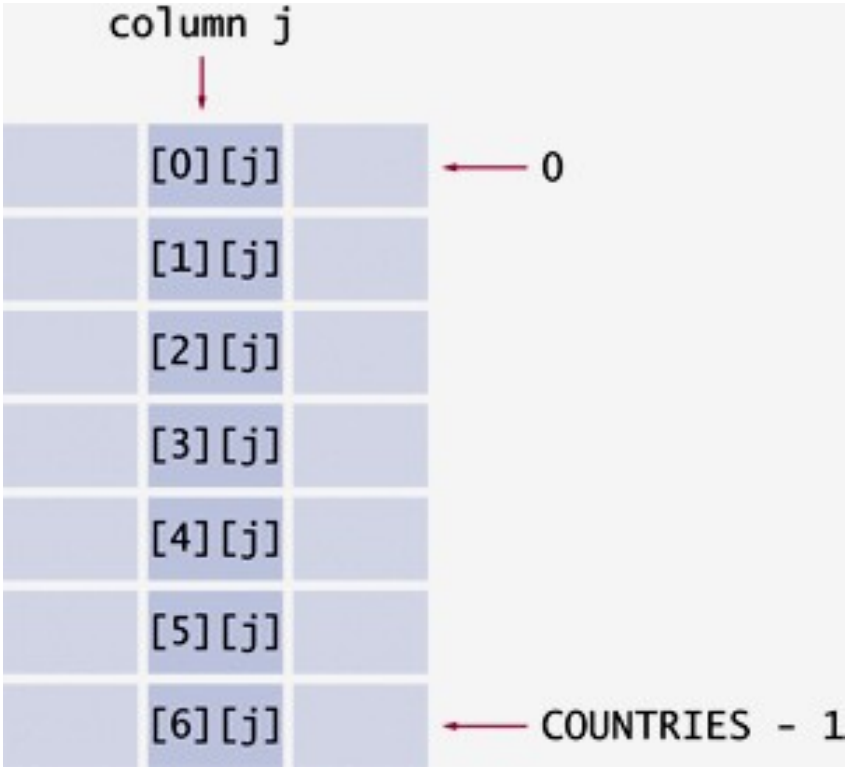
For each row `i`, we must use the column indices:

`0, 1, … (MEDALS -1)`

Computing Row and Column Totals

How many of each kind of medal (*metal!*) was won by the set of these particular countries?

counts



That would be a column total.

Let `j` be the silver column:

```
int total = 0;
for (int i = 0; i < COUNTRIES; i++) {
   total = total + counts[i][j];
}
```

Two-Dimensional Array Parameters

When passing a two-dimensional array to a function, you must specify the number of columns *as a constant* when you write the parameter type.

```
table[][COLUMNS]
```

Two-Dimensional Array Parameters

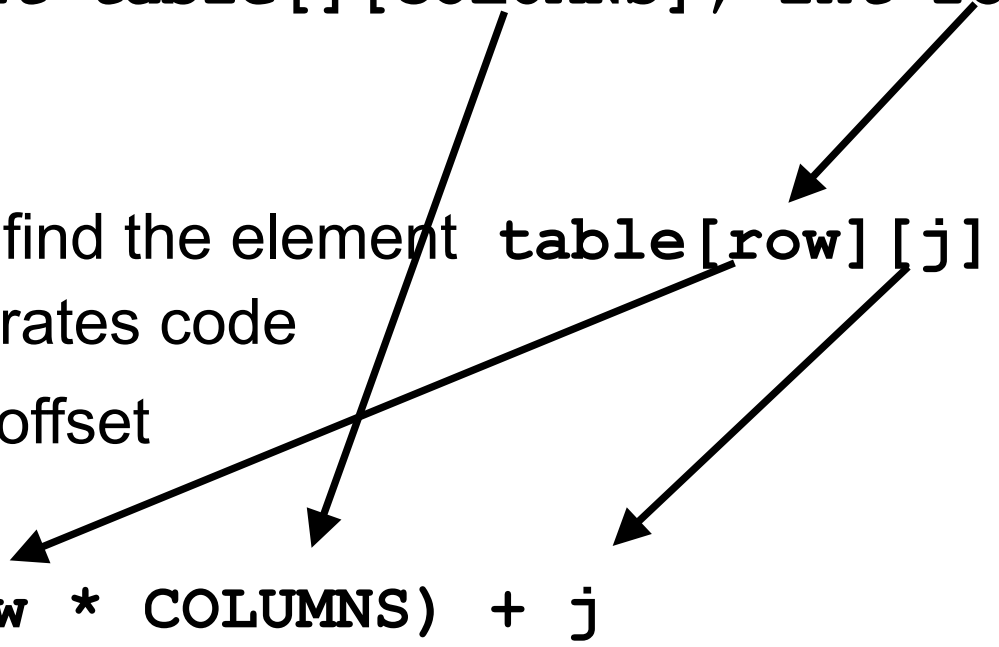
This function computes the total of a given row.

```
const int COLUMNS = 3;
int row_total(int table[][COLUMNS], int row)
{
   int total = 0;
   for (int j = 0; j < COLUMNS; j++) {
      total = total + table[row][j];
   }
   return total;
}
```

Two-Dimensional Array Parameters

```
int row_total(int table[][COLUMNS], int row)
```

In this function, to find the element `table[row][j]`
the compiler generates code
by computing the offset



```
(row * COLUMNS) + j
```

Two-Dimensional Array Parameters

That function works for only arrays of 3 columns.

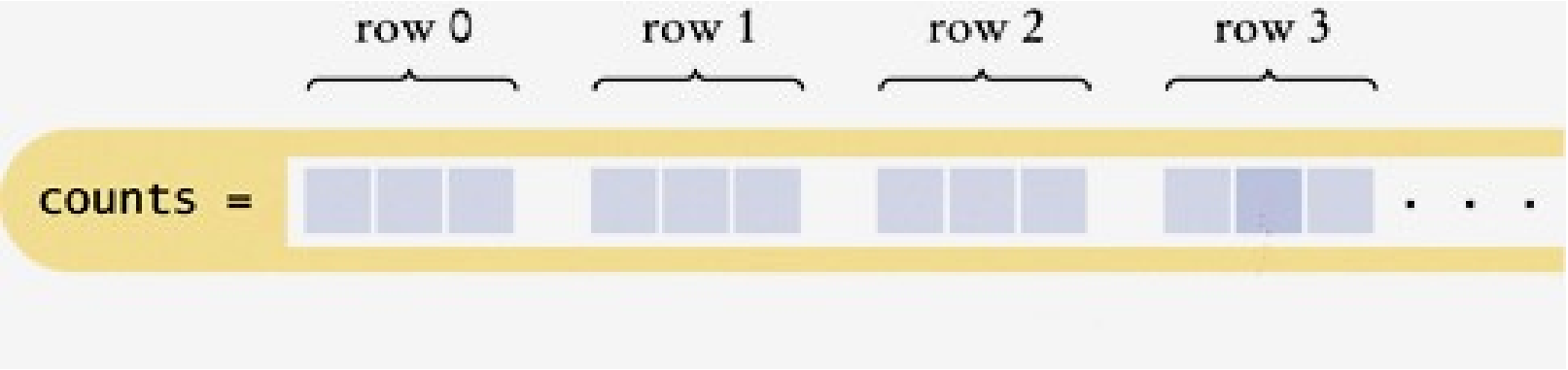
If you need to process an array
with a different number of columns, like 4,

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

Two-Dimensional Array Parameters

What's the reason behind this?

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



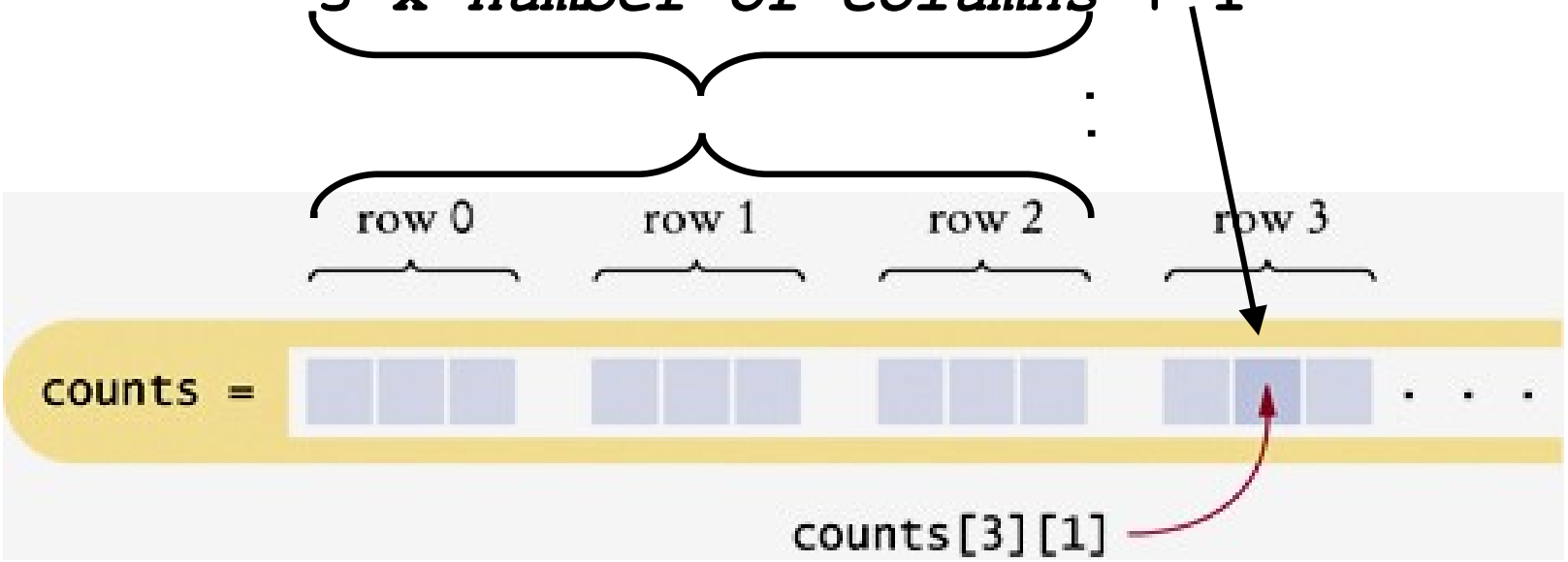
Two-Dimensional Array Parameters

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

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

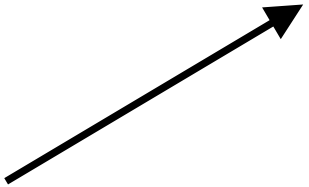
The offset from the start of the array is

`3 x number of columns + 1`



Two-Dimensional Array Parameters

```
int row_total(int table[][COLUMNS], int row)
```



`table[]` looks like a normal 1D array.

Notice the empty square brackets.

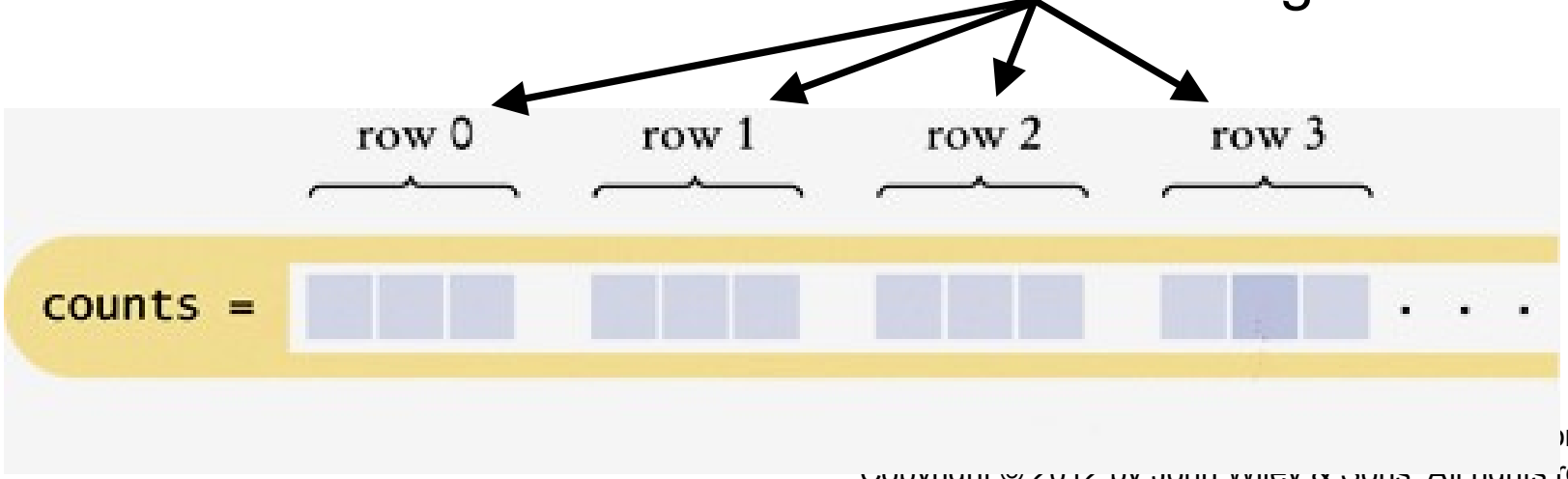
Two-Dimensional Array Parameters

```
int row_total(int table[][COLUMNS], int row)
```

`table[]` looks like a normal 1D array.

It is!

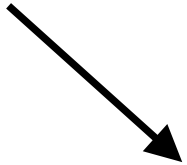
Each element is `COLUMNS ints` long.



Two-Dimensional Array Parameters

The `row_total` function did not need to know the number of rows of the array.

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;
}
```

Two-Dimensional Array Parameters – Common Error

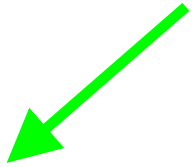
Leaving out the columns value is a very common error.

```
int row_total(int table[][], int row)
...
```

The compiler doesn't know how "long" each row is!

Two-Dimensional Array Parameters – Not an Error

Putting a value for the rows is not an error.

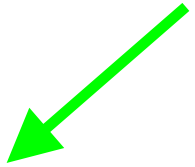


```
int row_total(int table[17][COLUMNS], int row)
...
```

The compiler just ignores whatever you place there.

Two-Dimensional Array Parameters – Not an Error

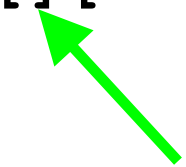
Putting a value for the rows is not an error.



```
int row_total(int table[17][COLUMNS], int row)
...
```

The compiler just ignores whatever you place there.

```
int row_total(int table[][COLUMNS], int row)
...
```



Never mind

Two-Dimensional Array Parameters

Here is the complete program for medal and column counts.

```
#include <stdio.h>
#include <stdlib.h>

const int COLUMNS = 3;
```

Two-Dimensional Array Parameters

ch06/medals.cpp

```
/**
 * Computes the total of a row in a table.
 * @param table a table with 3 columns
 * @param row the row that needs to be totaled
 * @return the sum of all elements in the given row
 */
double row_total(int table[][COLUMNS], int row)
{
   int total = 0;
   for (int j = 0; j < COLUMNS; j++) {
      total = total + table[row][j];
   }
   return total;
}
```

Two-Dimensional Array Parameters

ch06/medals.cpp

```
int main()
{
   const int COUNTRIES = 7;
   const int MEDALS = 3;

   char countries[][15] =
      {
         "Canada",
         "China",
         ”Germany",
         ”Korea",
         ”Japan",
         ”Russia",
         "United States"
      };
```

Two-Dimensional Array Parameters

ch06/medals.cpp

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

Two-Dimensional Array Parameters

```
   printf("    Country  Gold  Silver  Bronze   Total\n");

   // Print countries, counts, and row totals
   for (int i = 0; i < COUNTRIES; i++) {
      printf("%15d", countries[i]);
      // Process the ith row
      for (int j = 0; j < MEDALS; j++) {
         printf("%8d", counts[i][j]);
      }
      int total = row_total(counts, i);
      printf("%8d\n", total);
   }
   return EXIT_SUCCESS;
}
```

Arrays – One 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.

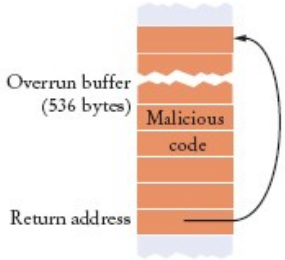
The compiler has to know the size to build it.
and a function must be told about the number elements and possibly the capacity.

It cannot hold more than it's initial capacity.

CHAPTER SUMMARY

Use arrays for collecting values.

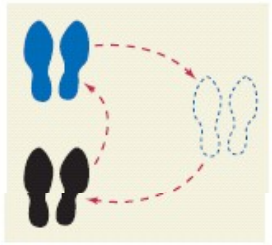
- Use an array to collect a sequence of values of the same type.
- Individual elements in an array *values* are accessed by an integer index `i`, using the notation *values*`[i]`.
- An array element can be used like any variable.
- An array index must be at least zero and less than the size of the array.
- A bounds error, which occurs if you supply an invalid array index, can corrupt data or cause your program to terminate.
- With a partially filled array, keep a companion variable for the current size.



CHAPTER SUMMARY

Be able to use common array algorithms.

- To copy an array, use a loop to copy its elements to a new array.
- When separating elements, don't place a separator before the first element.
- A linear search inspects elements in sequence until a match is found.
- Before inserting an element, move elements to the end of the array *starting with the last one*.
- Use a temporary variable when swapping two elements.



Implement functions that process arrays.

- When passing an array to a function, also pass the size of the array.
- Array parameters are always reference parameters.
- A function's return type cannot be an array.
- When a function modifies the size of an array, it needs to tell its caller.
- A function that adds elements to an array needs to know its capacity.

CHAPTER SUMMARY

Be able to combine and adapt algorithms for solving a programming problem.

- By combining fundamental algorithms, you can solve complex programming tasks.
- You should be familiar with the implementation of fundamental algorithms so that you can adapt them.

Discover algorithms by manipulating physical objects.

- Use a sequence of coins, playing cards, or toys to visualize an array of values.
- You can use paper clips as position markers or counters.

Use two-dimensional arrays for data that is arranged in rows and columns.

- Use a two-dimensional array to store tabular data.
- Individual elements in a two-dimensional array are accessed by using two subscripts, *array*`[i][j]`.
- A two-dimensional array parameter must have a fixed number of columns.

End Chapter Six