Chapter Six: Arrays and Vectors I

Slides by Evan Gallagher & Nikolay Kirov

Lecture 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

Using Arrays and Vectors

Mail, mail and more mail – how to manage it?

Using Vectors

- When you need to work with a large number of values – all together, the vector construct is your best choice.
- By using a ***vector*** you
  - can conveniently manage collections of data
  - do not worry about the details of how they are stored
  - do not worry about how many are in the vector
    - a vector automatically grows to any desired size

Using Arrays

- Arrays are a lower-level construct
- The ***array*** is
  - less convenient
  - but sometimes required
    - for efficiency
    - for compatibility with older software

Using Arrays and Vectors

All the mail these days seems alike: ***junk!***

Using Arrays and Vectors

In both vectors and arrays,
the stored data is of
the *same* type

Using Arrays and Vectors

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)

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 and Vectors

32 54 67.5 29 35 80 115 44.5 100 65

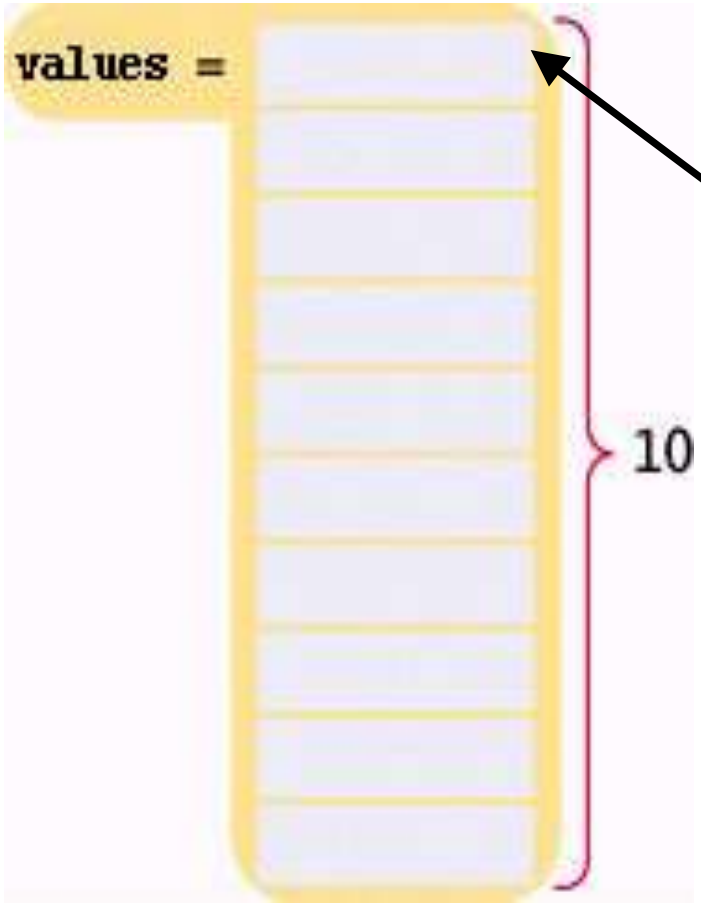
So you would create a variable for each, of course!

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

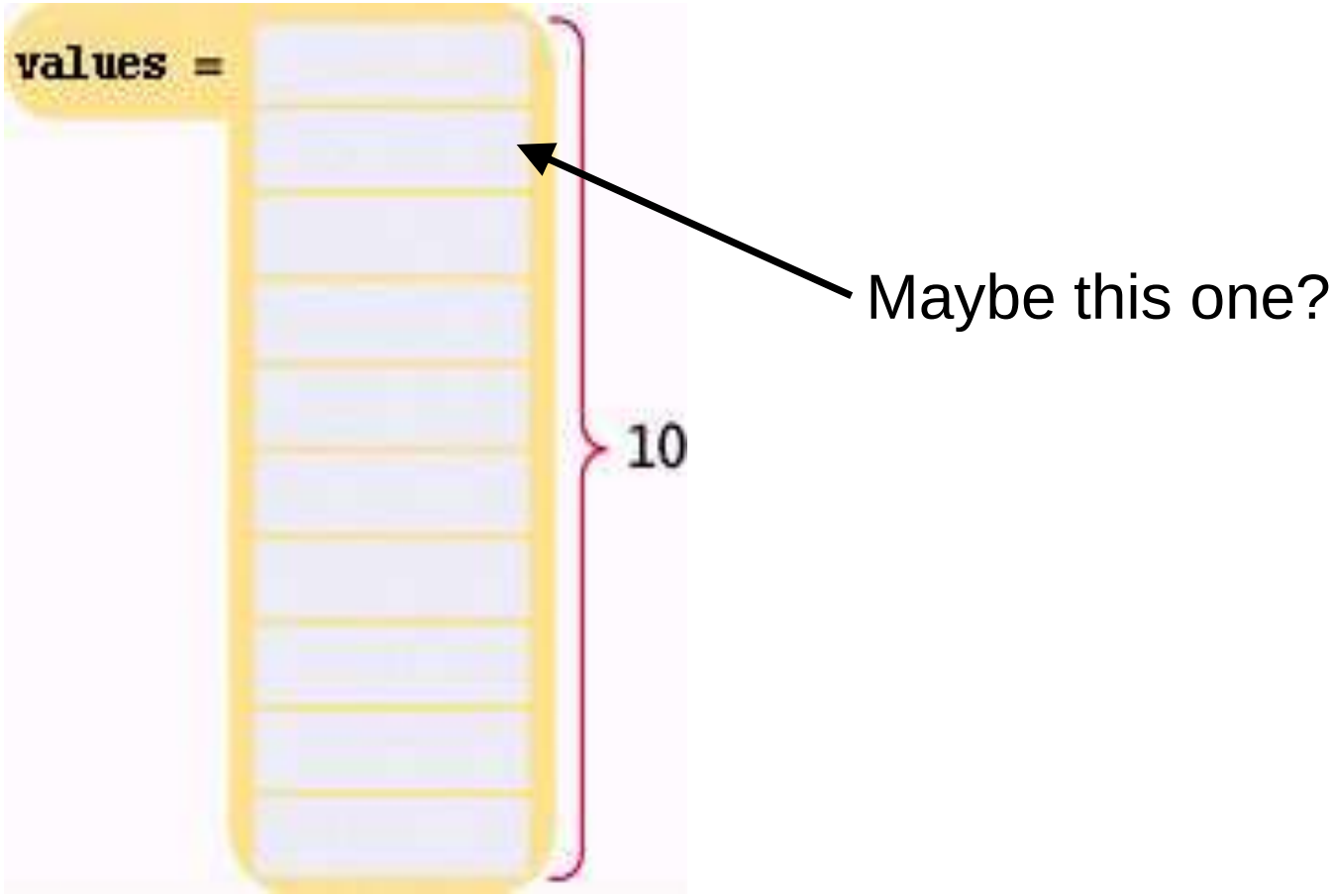
Then what ???

Using Arrays and Vectors

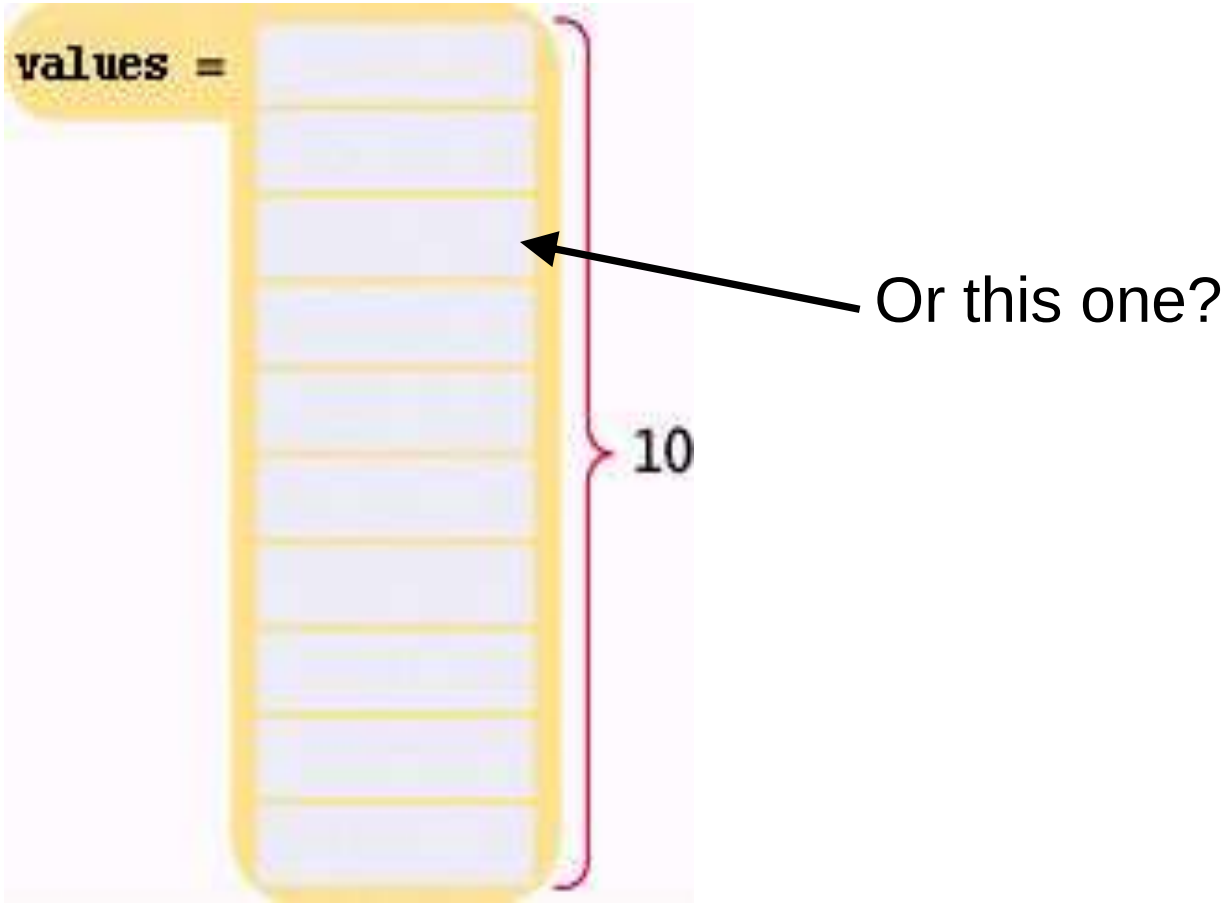
You can easily visit each element in an array or in a vector, checking and updating a variable holding the current maximum.



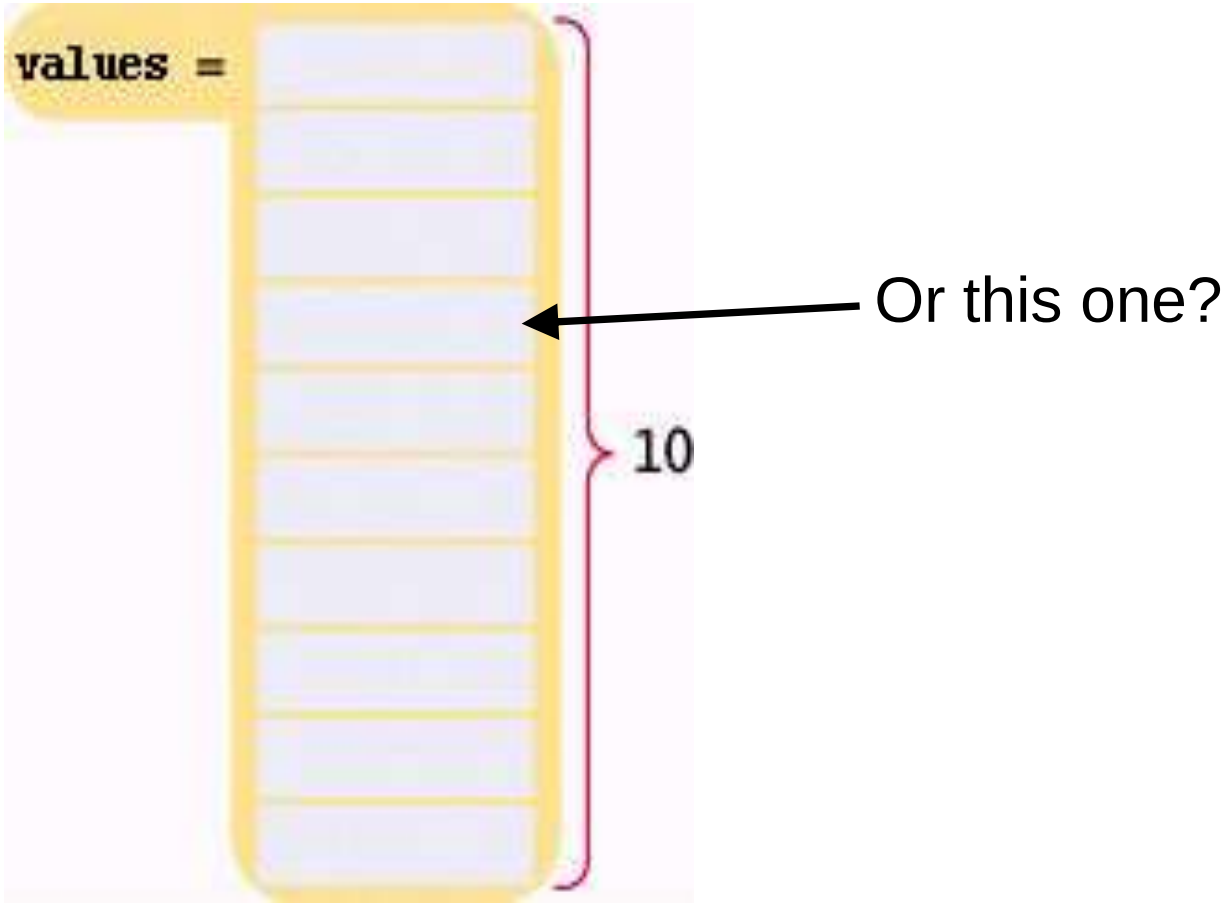
Using Arrays and Vectors



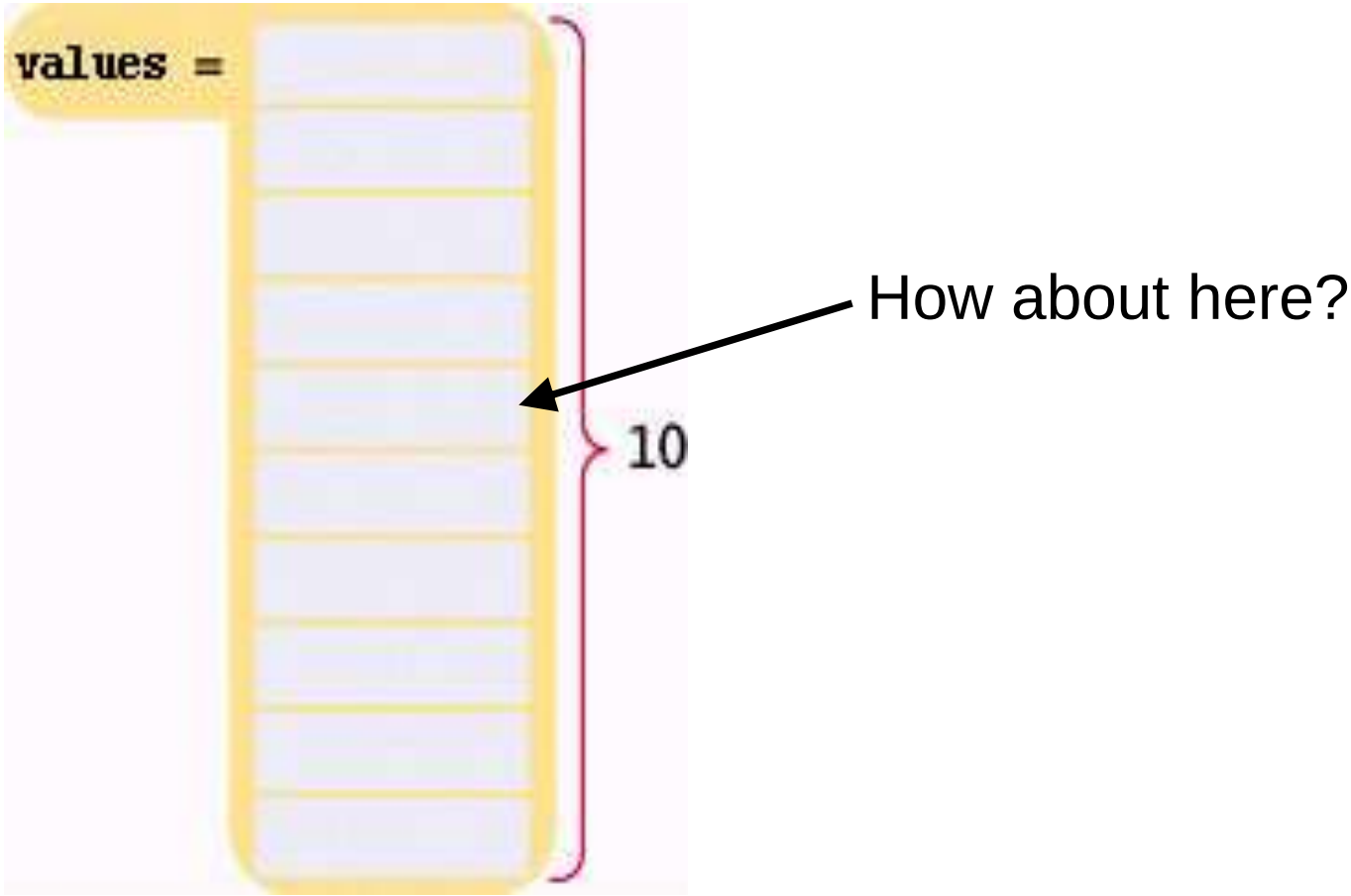
Using Arrays and Vectors



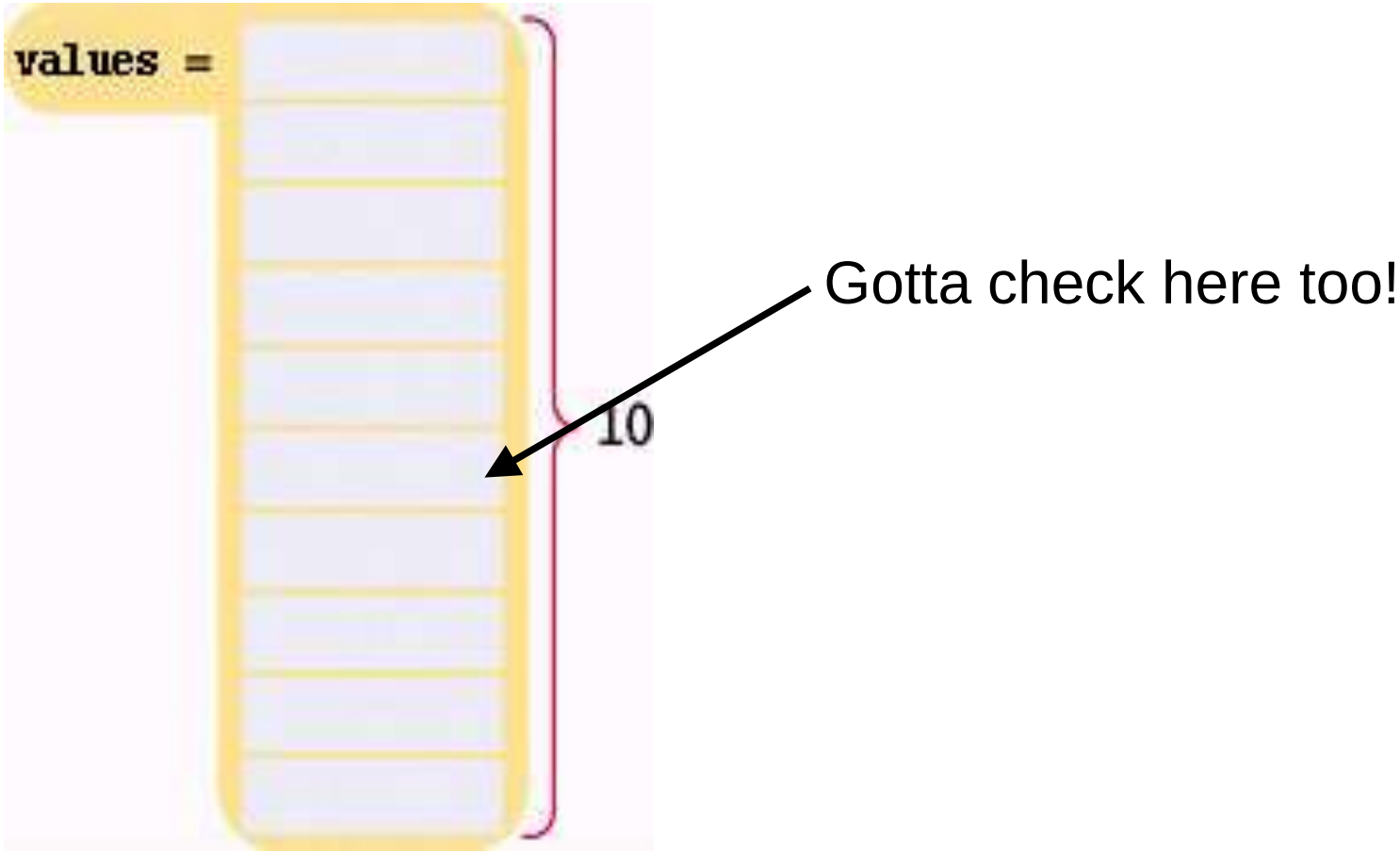
Using Arrays and Vectors



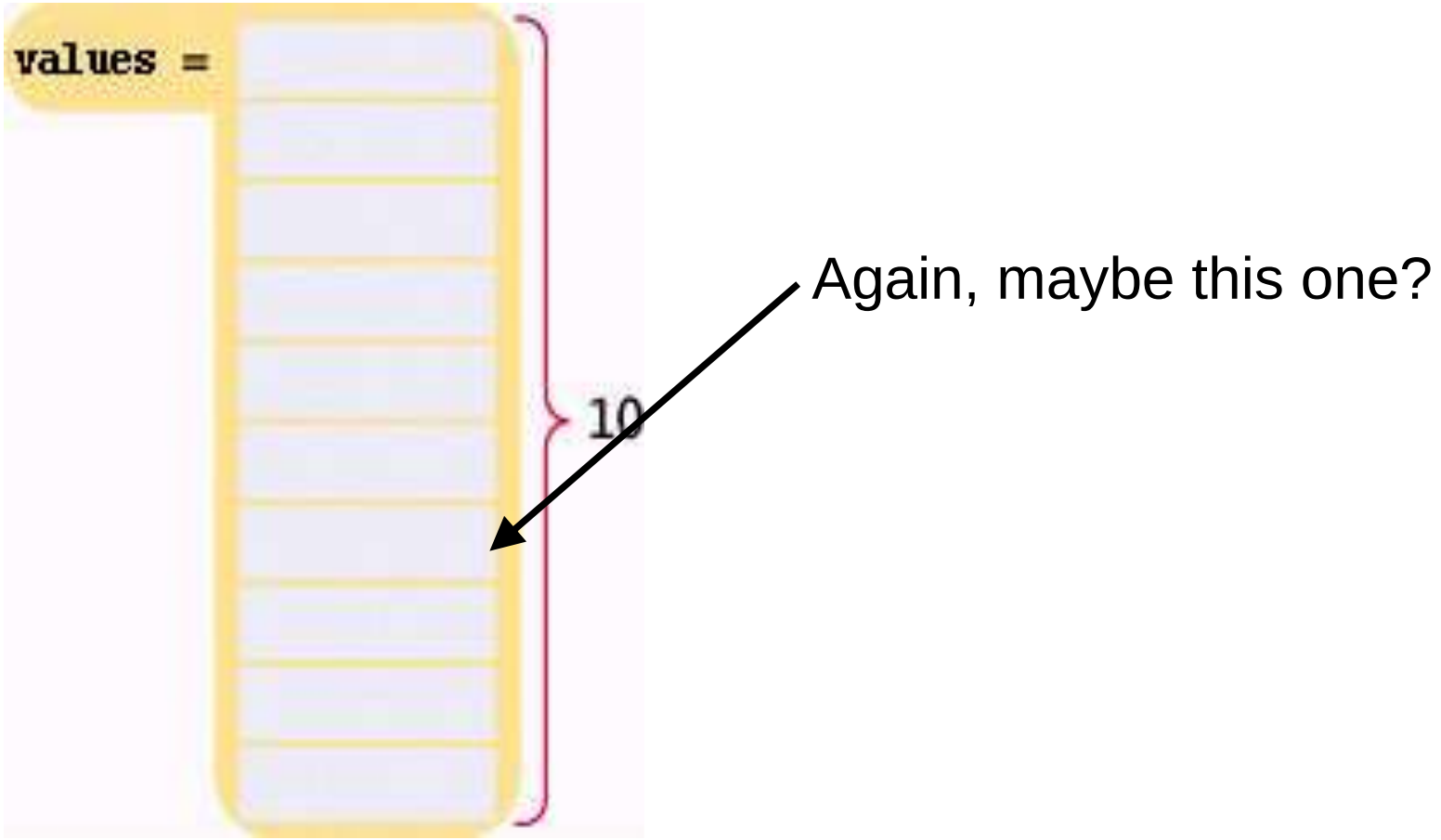
Using Arrays and Vectors



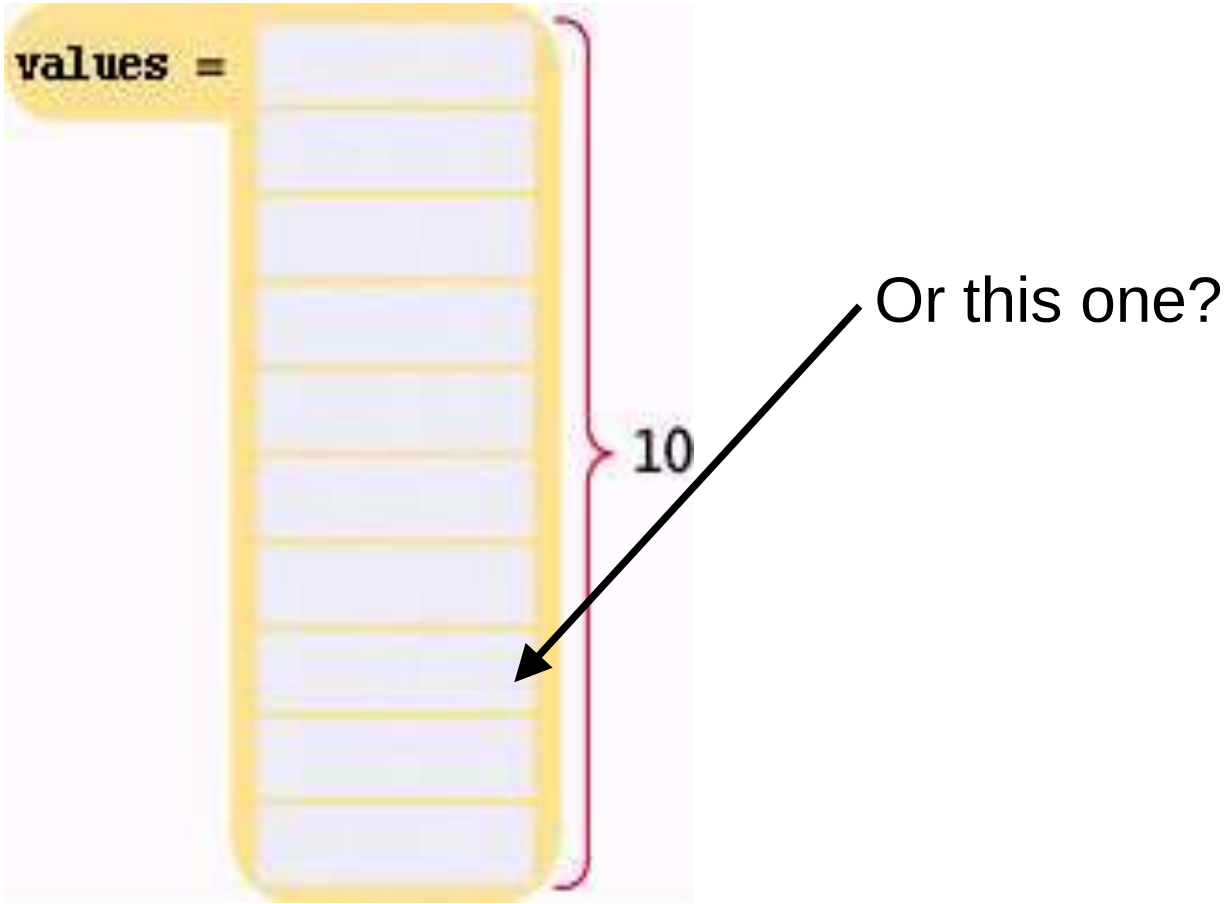
Using Arrays and Vectors



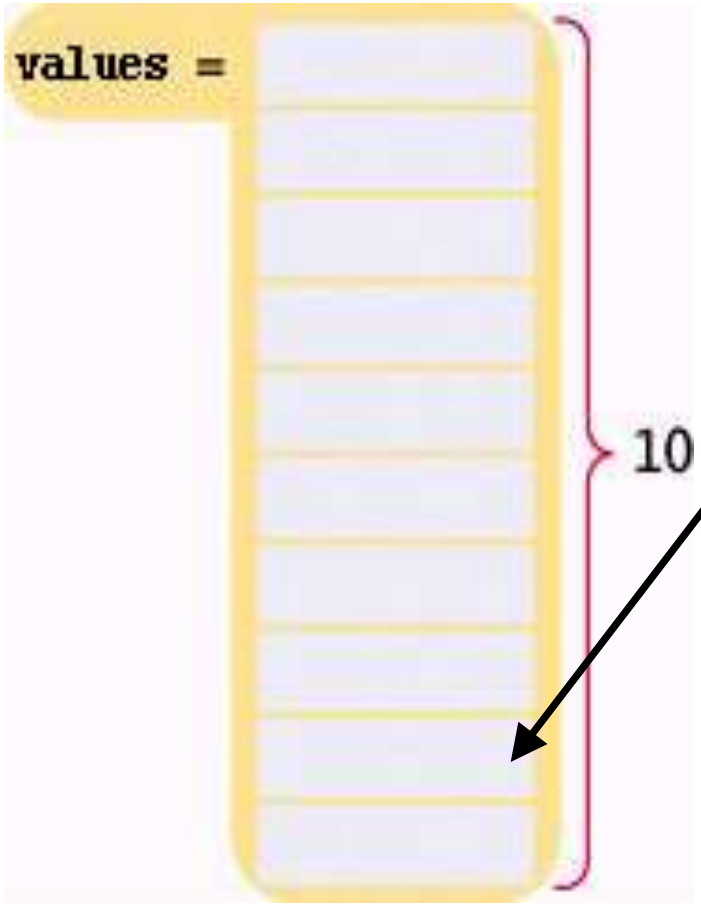
Using Arrays and Vectors



Using Arrays and Vectors



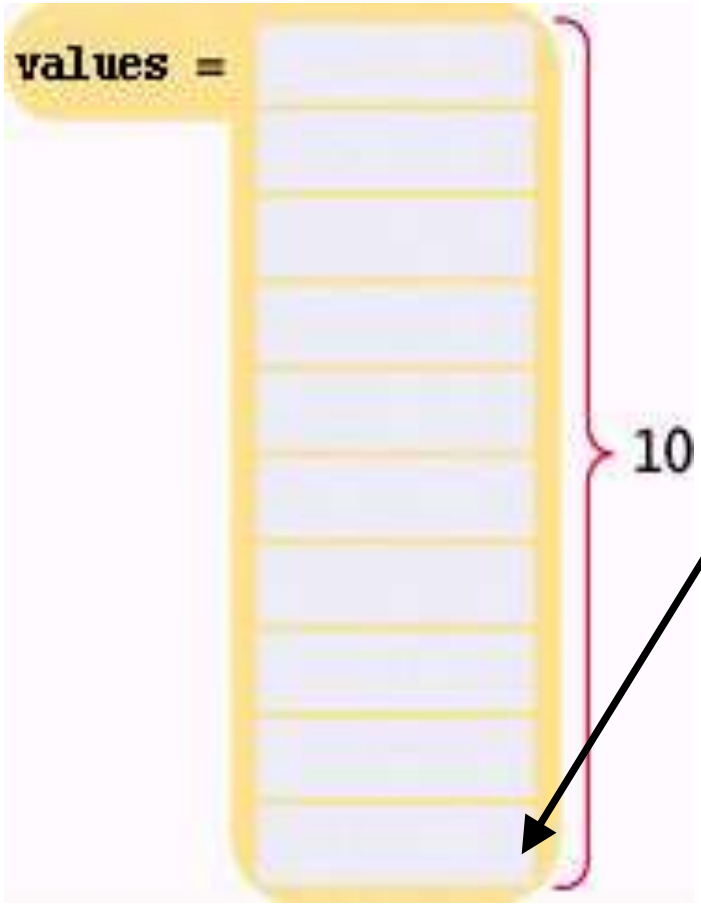
Using Arrays and Vectors



Or this one?

Will this never end!

Using Arrays and Vectors



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

Using Arrays and Vectors

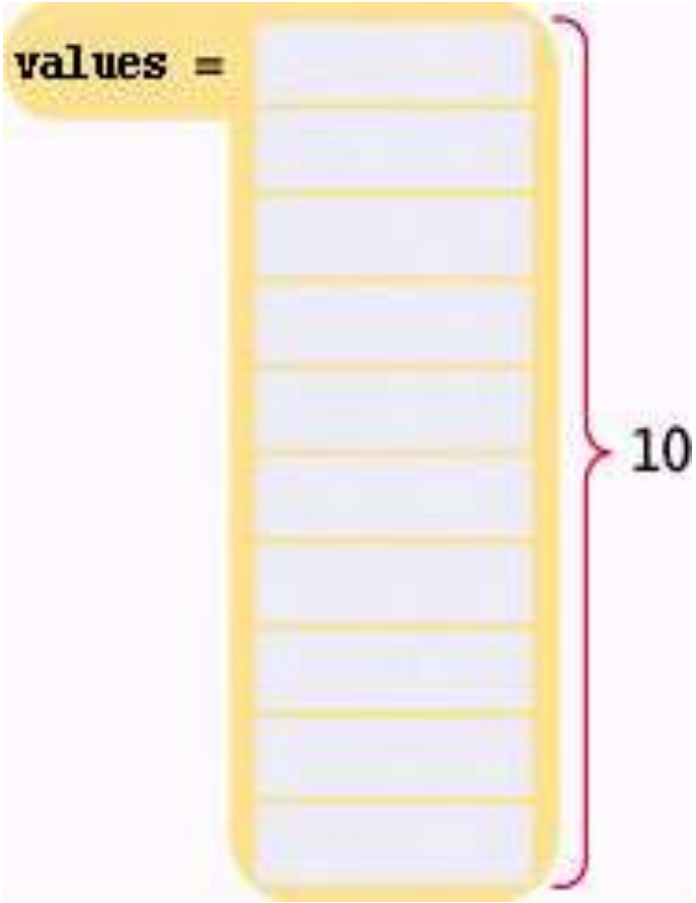
That would have been impossible with ten separate variables!

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

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

ARGH!

Defining Arrays



An “array of double”

Ten elements of **double** type can be stored under one name as an array.

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

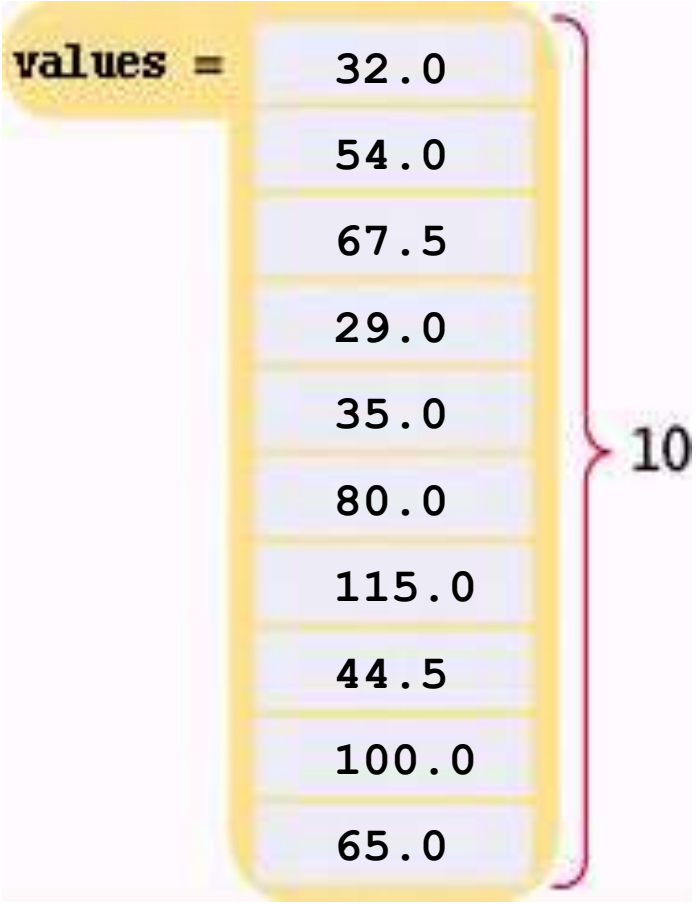
type of each element

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 };
```



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*.

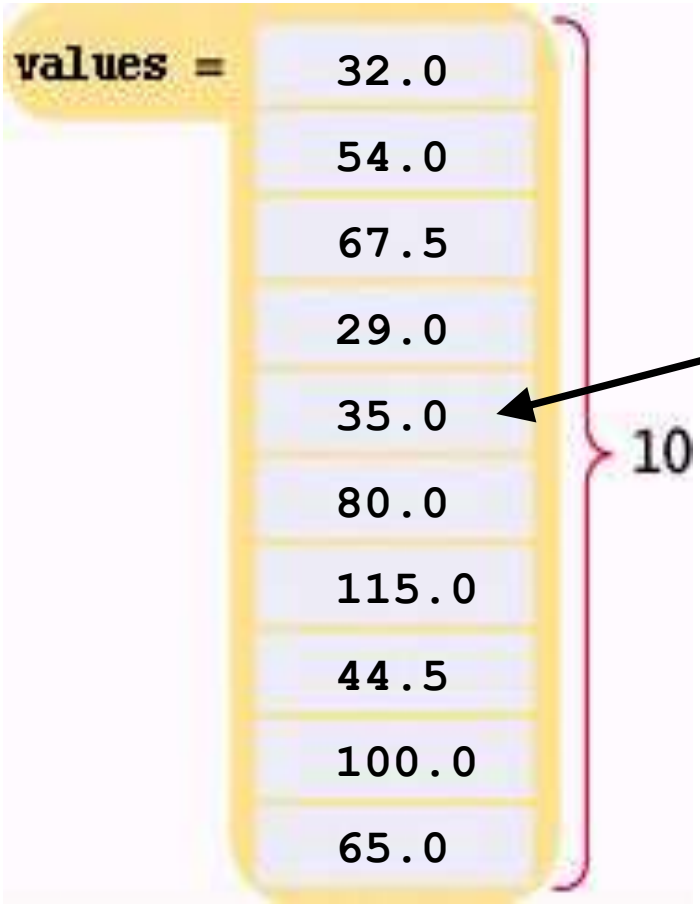
Accessing an Array Element

Put the junk mail in there

in **`mailboxes[356]`**

Accessing an Array Element

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

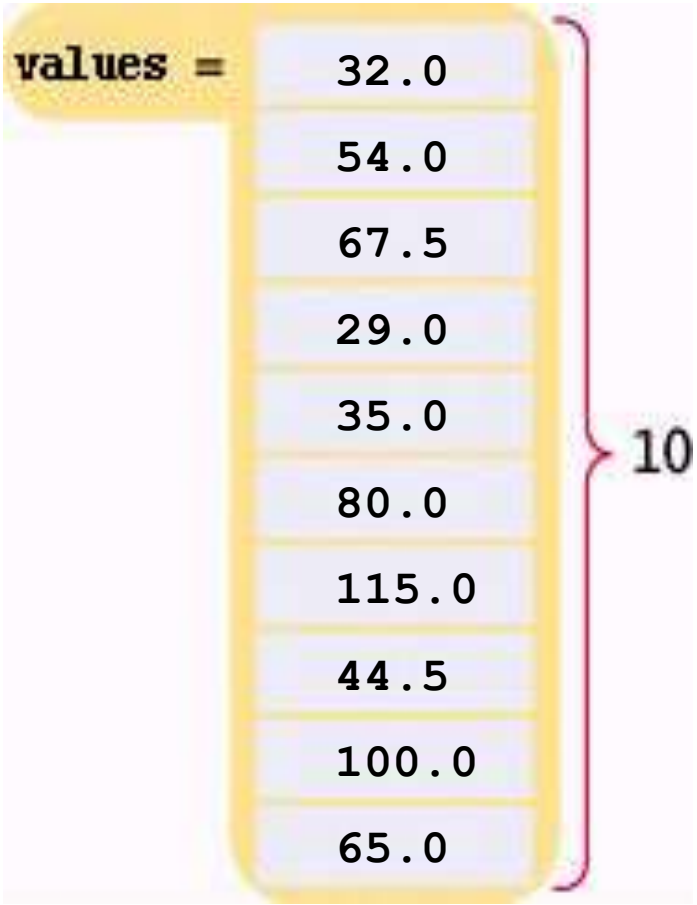



```
double values[10];
...
cout << values[4] << endl;
```

The output will be **`35.0`**.

Accessing an Array Element

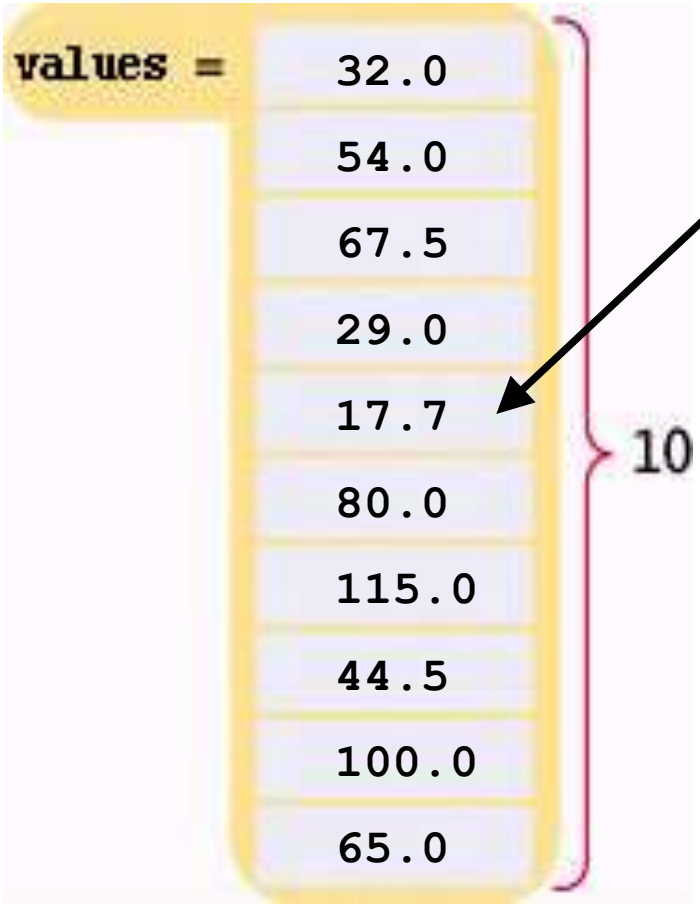
The same notation can be used to change the element.




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

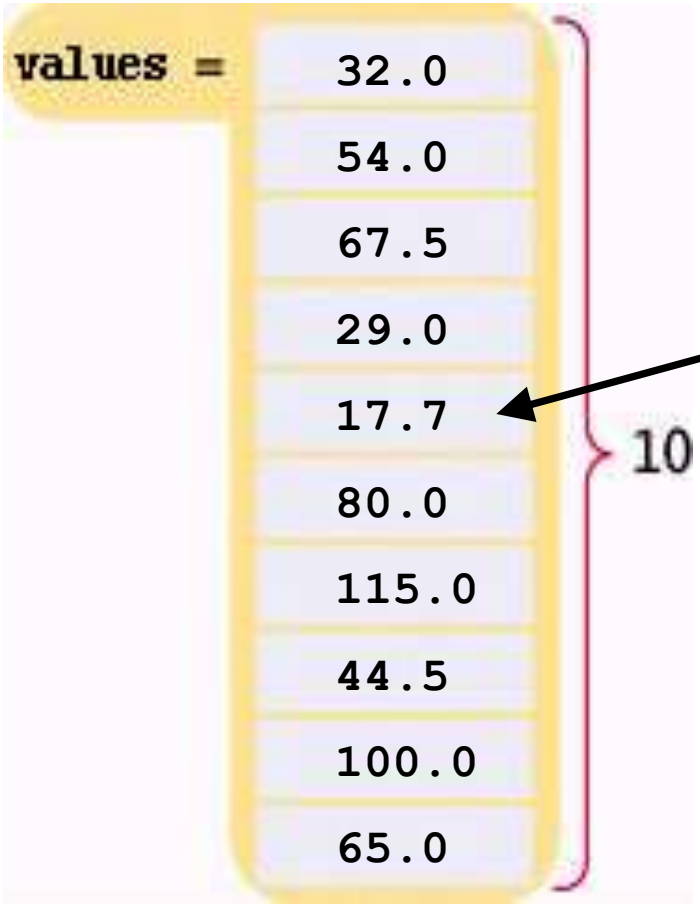
Accessing an Array Element

The same notation can be used to change the element.



Accessing an Array Element

The same notation can be used to change the element.

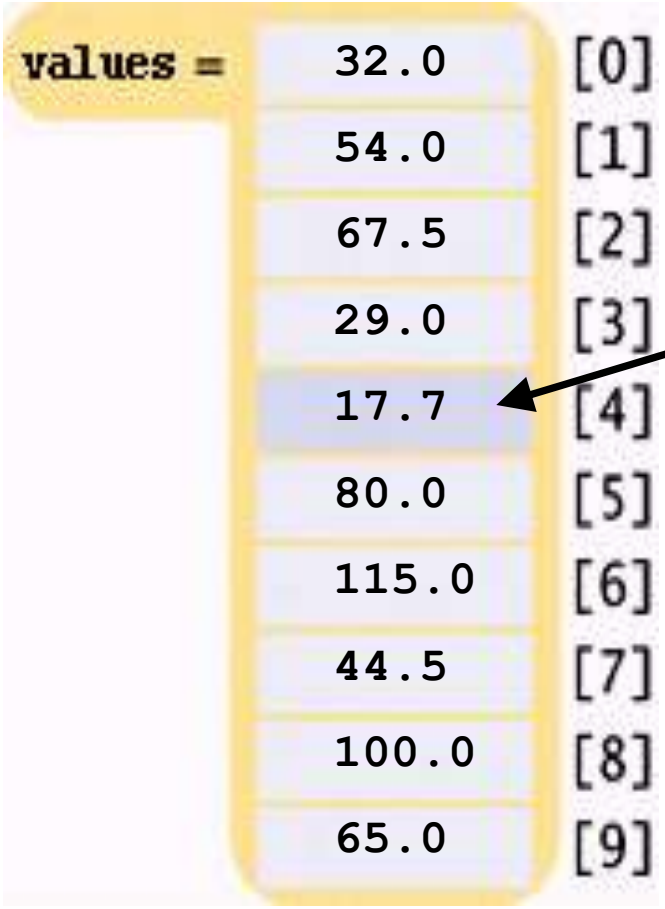



```
values[4] = 17.7;
cout << values[4] << endl;
```

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.




```
cout << values[4] << endl;
```

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

Accessing an Array Element

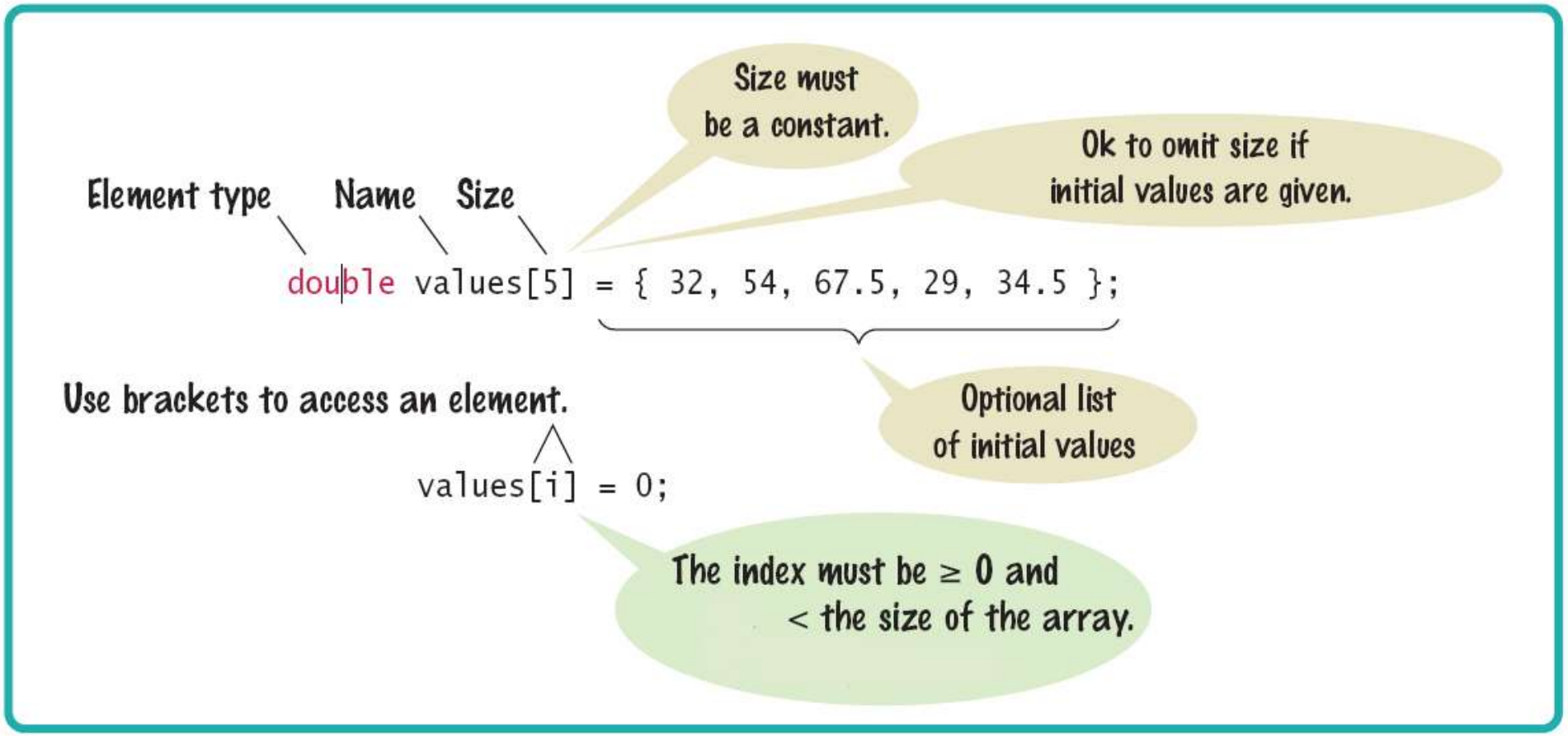
That is, the legal elements for the **`values`** array are:

`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.

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.

Partially-Filled Arrays

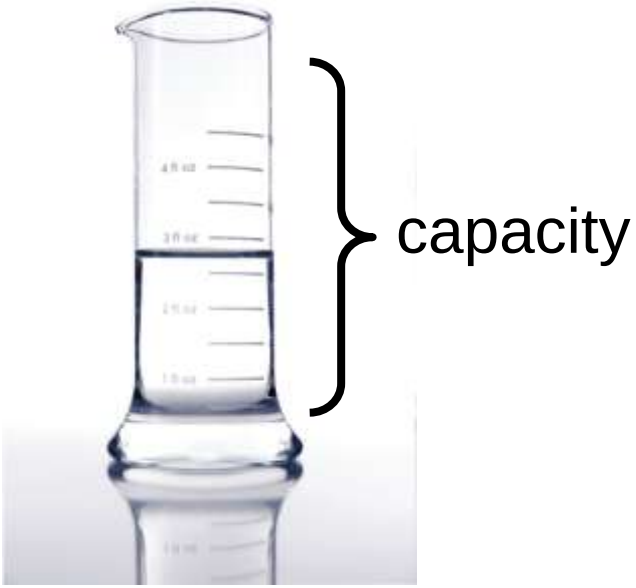
Suppose an array can hold 10 elements:

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`**:

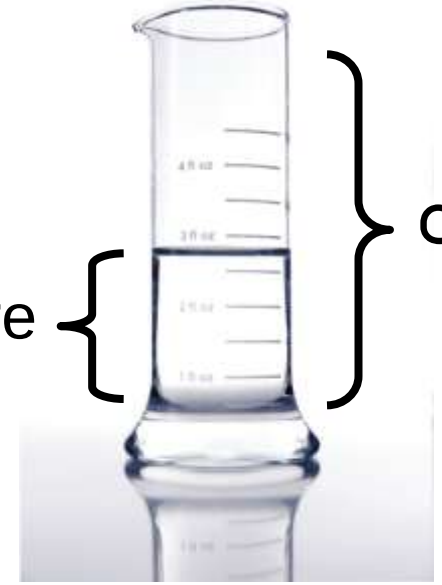
```
const int CAPACITY = 100;
double values[CAPACITY];
```

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

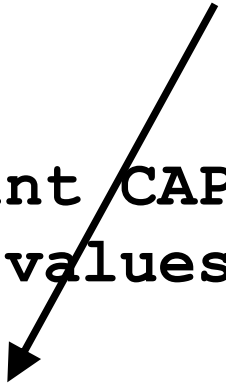


`CAPACITY`

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:

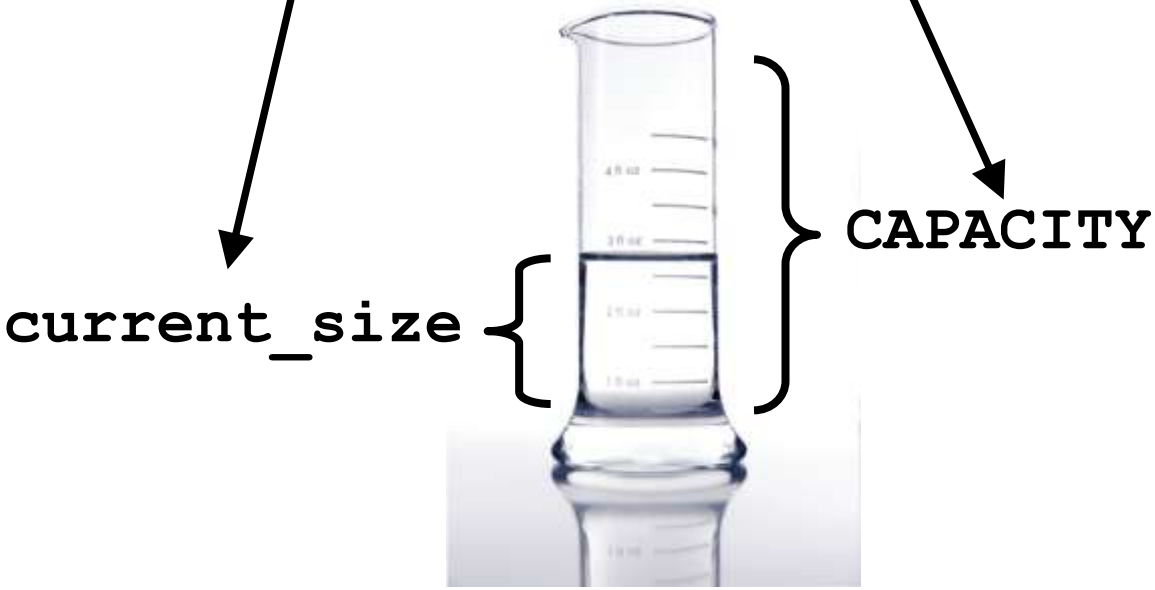


```
const int CAPACITY = 100;
double values[CAPACITY];

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

Suppose we add four elements to the array?

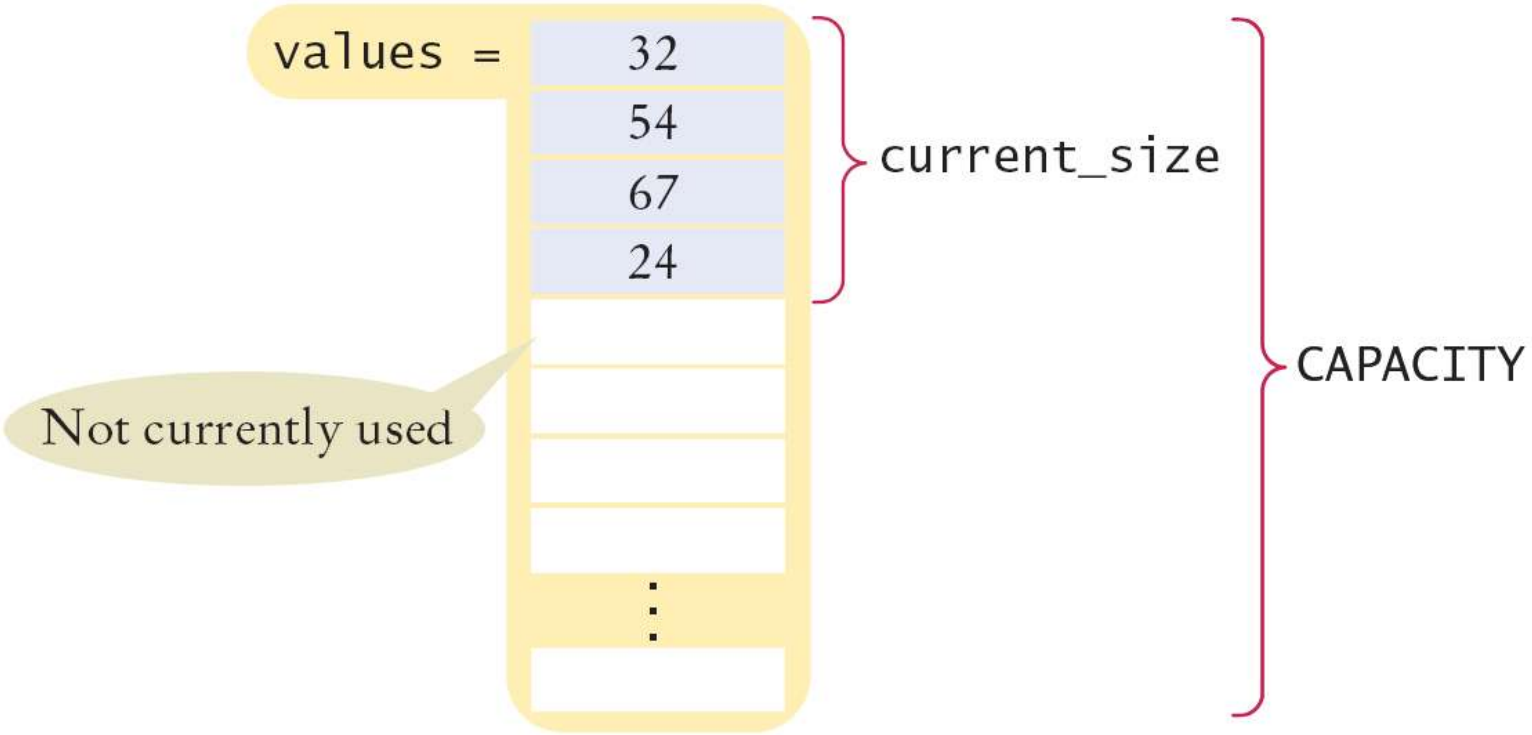
Partially-Filled Arrays – Companion Variable for Size

```
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.

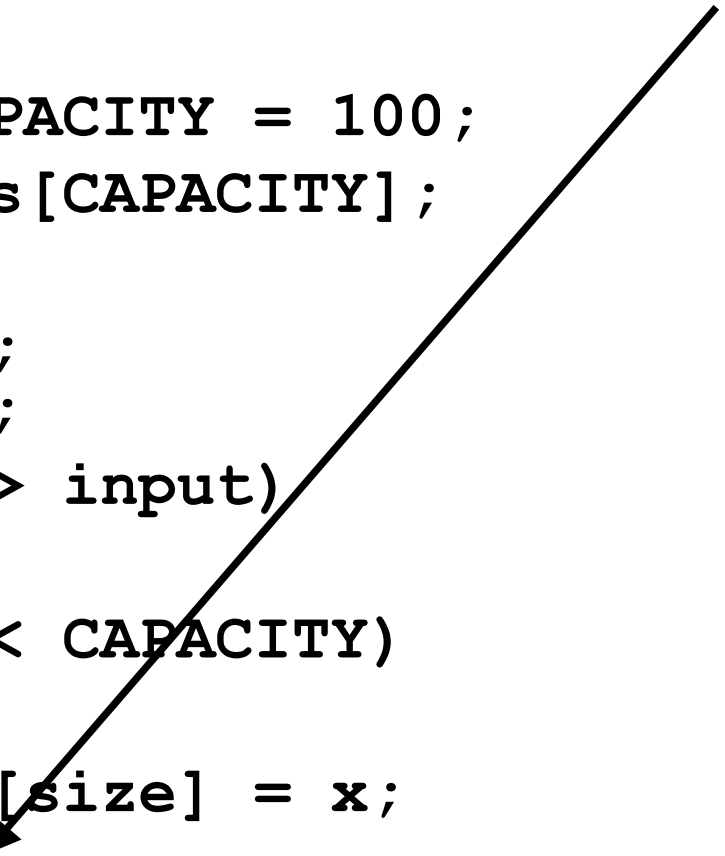
Each time the size of the array changes we update this variable:

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

int size = 0;
double input;
while (cin >> input)
{
   if (size < CAPACITY)
   {
      values[size] = x;
      size++;
   }
}
```

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;
while (cin >> input)
{
   if (size < CAPACITY)
   {
      values[size] = x;
      size++;
   }
}
```

Partially-Filled Arrays – Capacity

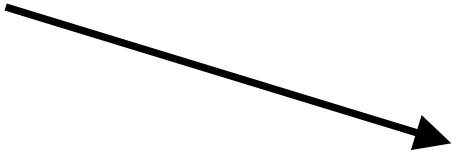
When the loop ends, the companion variable **size** has the number of elements in the array.

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

int size = 0;
double input;
while (cin >> input)
{
   if (size < CAPACITY)
   {
      values[size] = x;
      size++;
   }
}
```

Partially-Filled Arrays – Capacity

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++)
{
   cout << values[i] << endl;
}
```

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 < CAPACITY; i++)
{
   cout << values[i] << endl;
}
```

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 < CAPACITY; i++)
{
   cout << values[i] << endl;
}
```

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 < CAPACITY; i++)
{
   cout << values[0] << endl;
}
```

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 < CAPACITY; i++)
{
   cout << values[i] << endl;
}
```

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 < CAPACITY; i++)
{
   cout << values[1] << endl;
}
```

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 < CAPACITY; i++)
{
   cout << values[i] << endl;
}
```

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 < CAPACITY; i++)
{
   cout << values[2] << endl;
}
```

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 < CAPACITY; i++)
{
   cout << values[i] << endl;
}
```

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 < CAPACITY; i++)
{
   cout << values[9] << endl;
}
```

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 < CAPACITY; i++)
{
   cout << values[i] << endl;
}
```

Note that the loop condition is that the index is

***less than* `CAPACITY`**

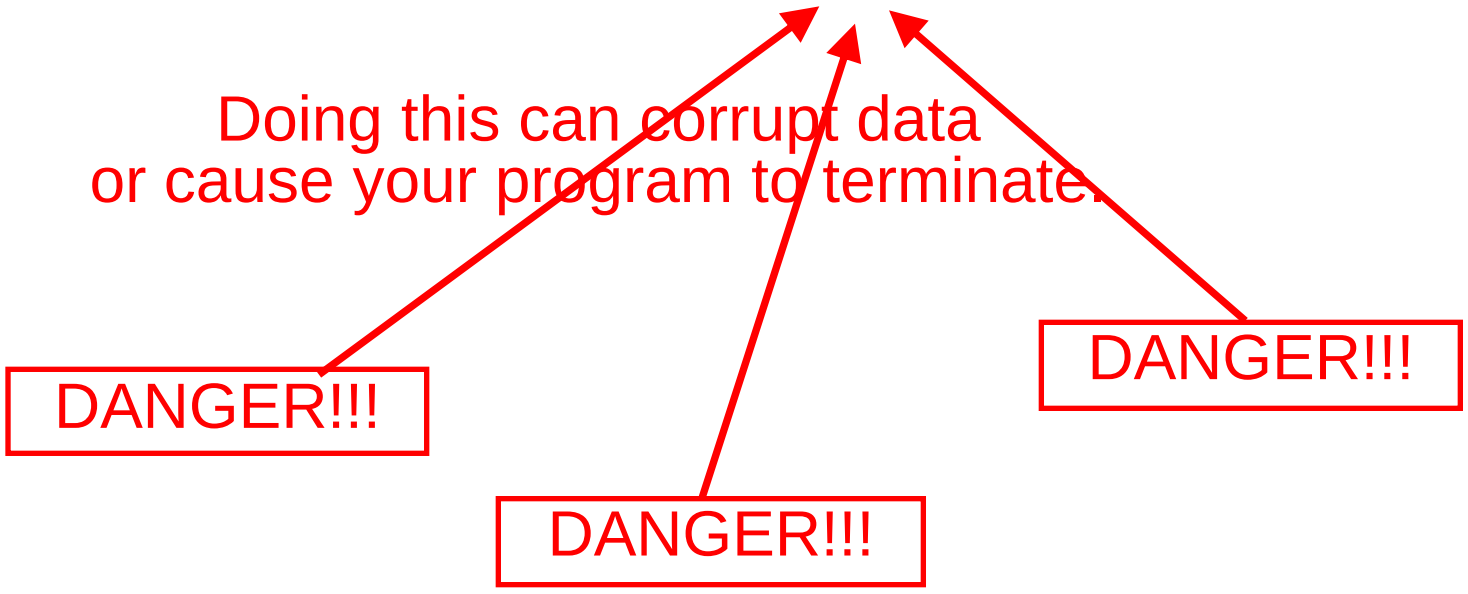
because there is no element corresponding to **`values`** [10].

But **`CAPACITY`** (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:

```
cout << 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.

(even if it is a bad score, it’s still 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 **`double`**s 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

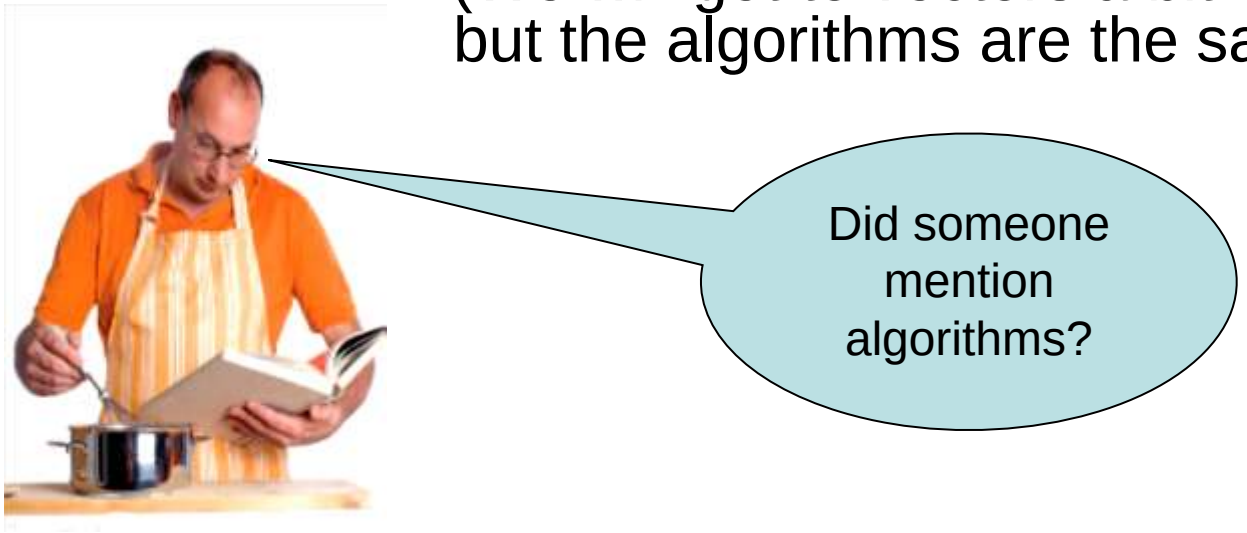
the meaning of each element is the same.

Common Array and Vector Algorithms

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

There many common algorithms for processing values stored in both arrays and vectors.

(We will get to vectors a bit later but the algorithms are the same)



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*!

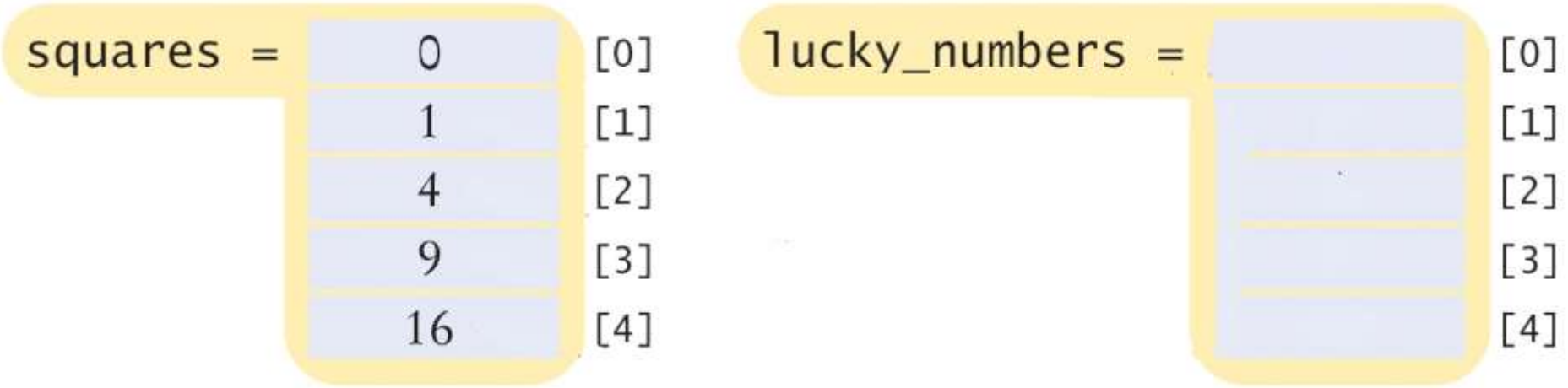
You cannot assign arrays!

You will have to do your own work, son.

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];
}
```

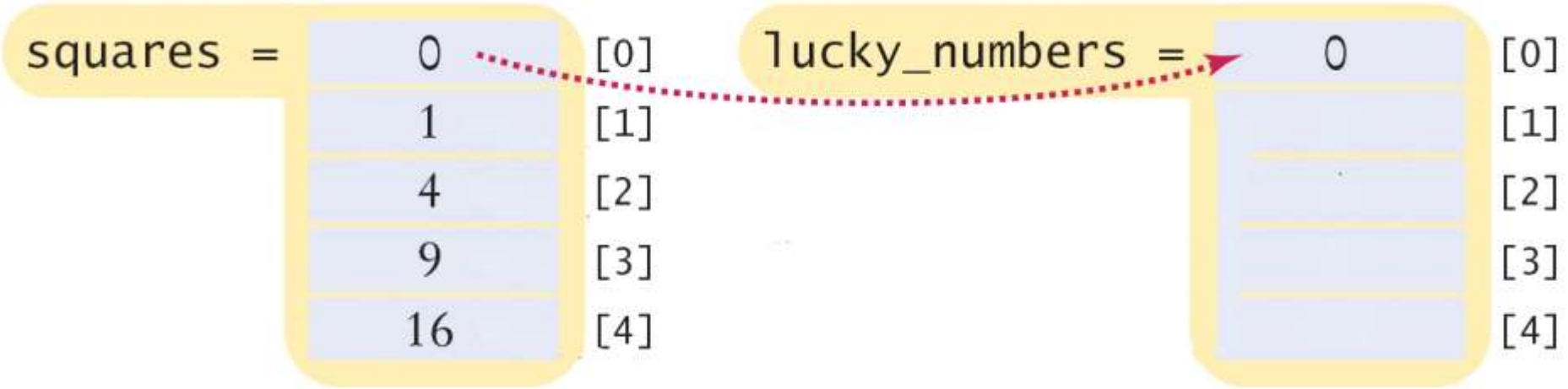
when `i` is `0`



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];
}
```

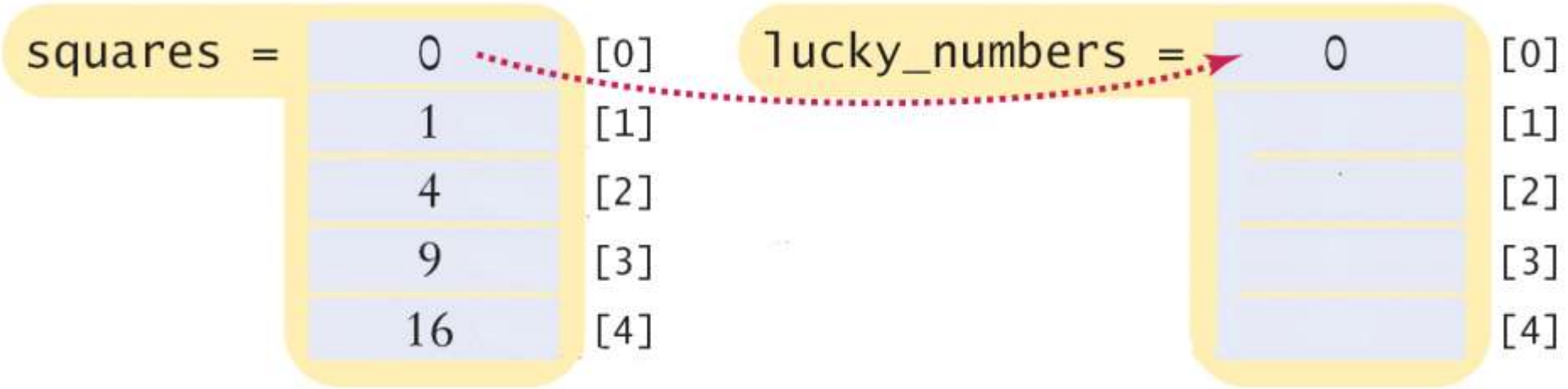
when `i` is `0`



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];
}
```

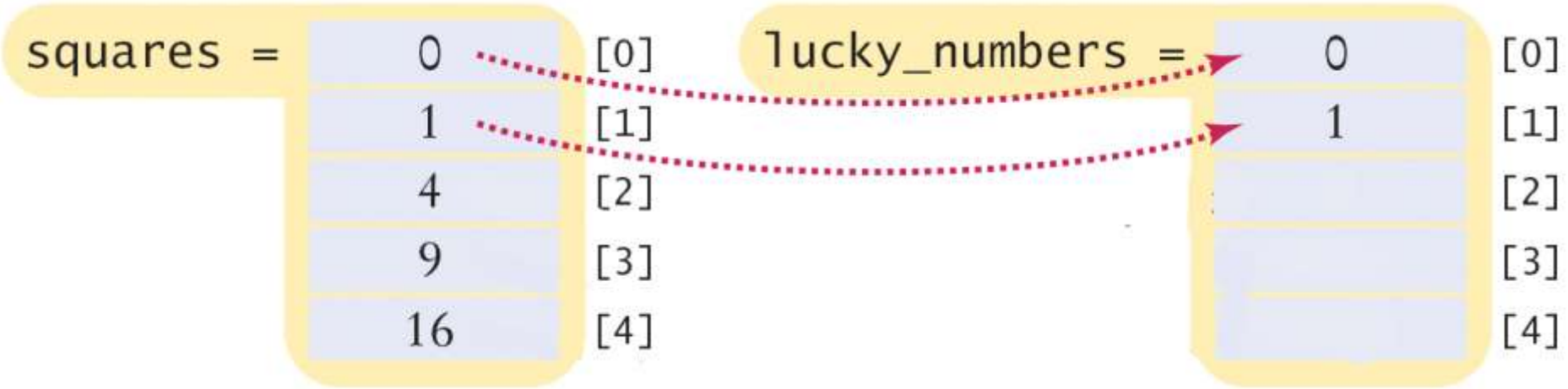
when `i` is `1`



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];
}
```

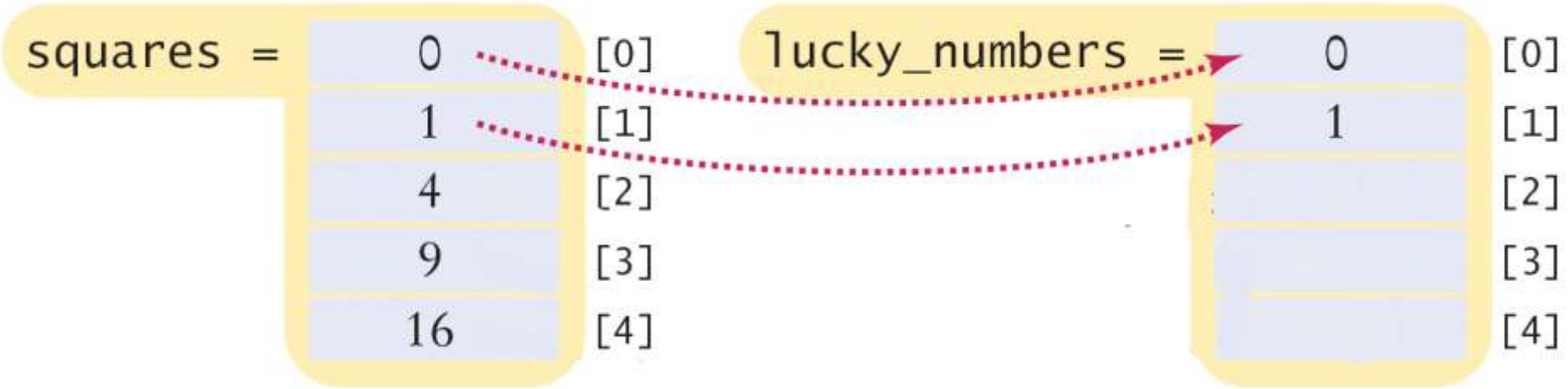
when `i` is `1`



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];
}
```

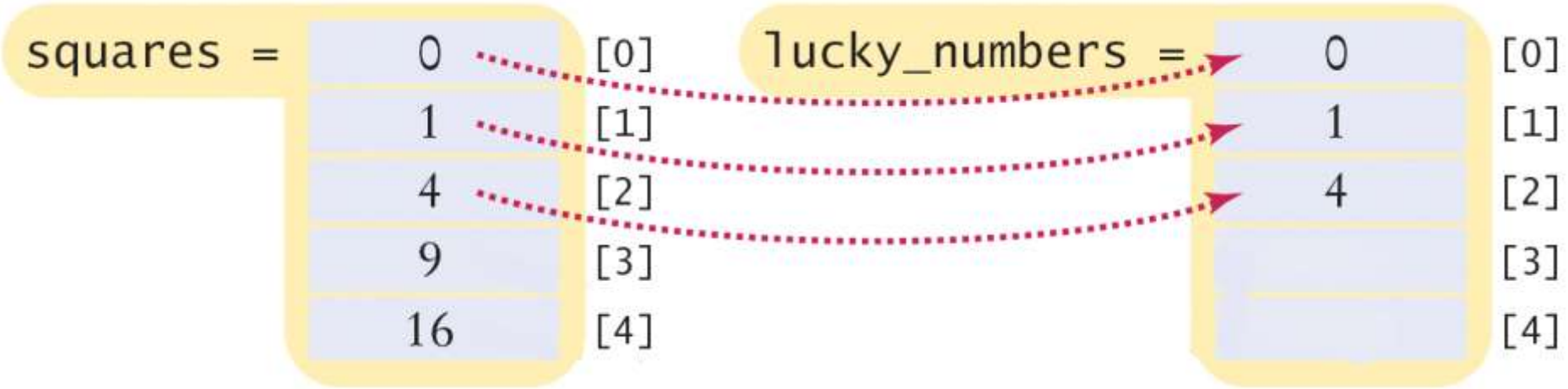
when `i` is `2`



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];
}
```

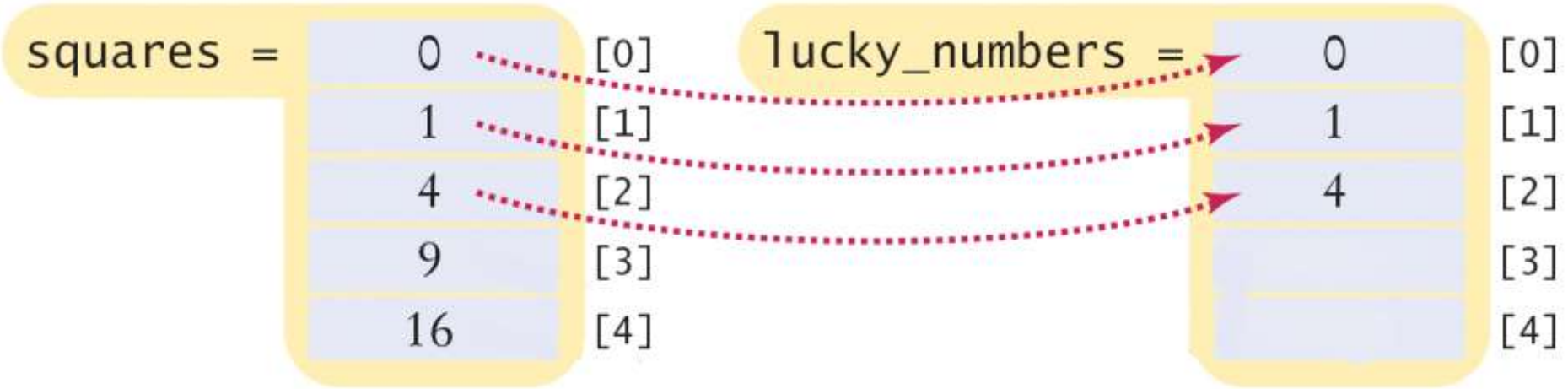
when `i` is `2`



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];
}
```

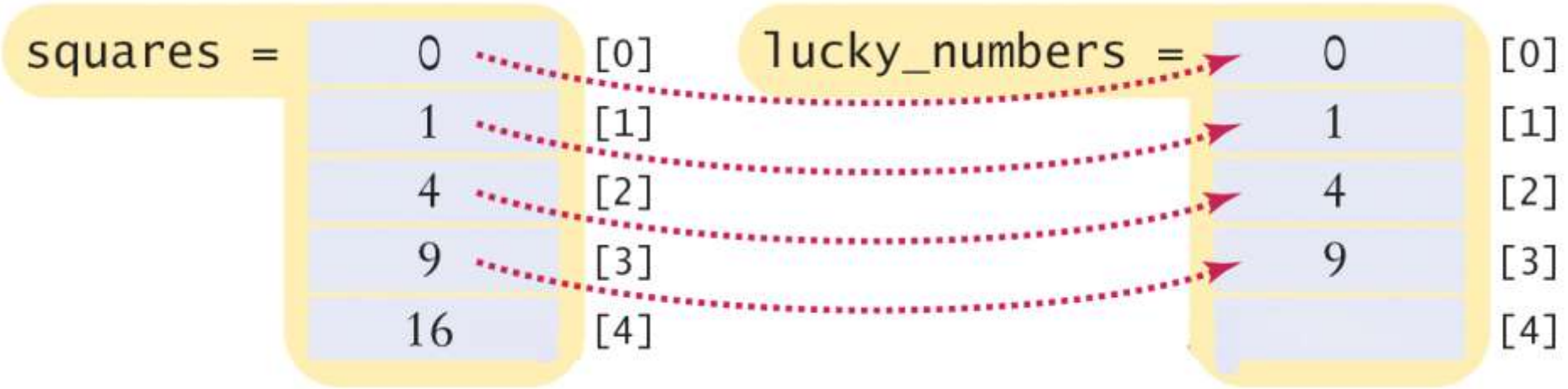
when `i` is `3`



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];
}
```

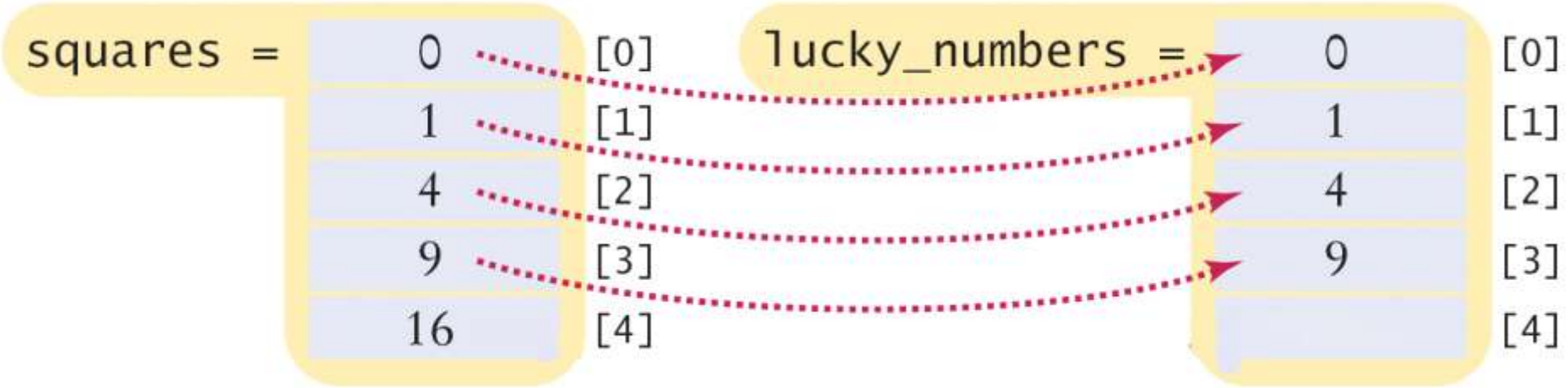
when `i` is `3`



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];
}
```

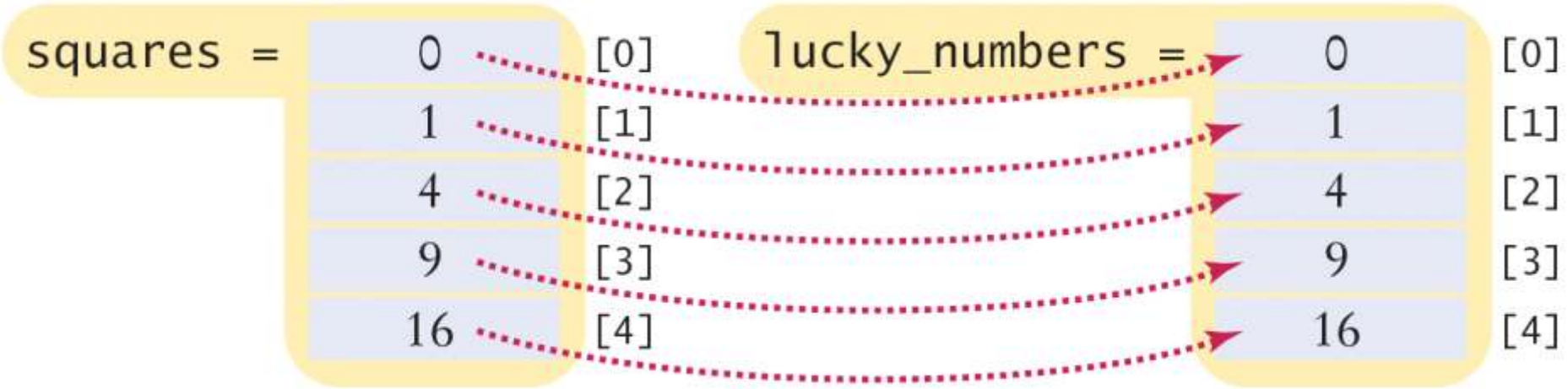
when `i` is `4`



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
for computing the su ta.
The algorithm is the ed in an array.

Same algorithm
- different situation.

Aren't algorithms great!

```
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 – Maximum and Minimum

To compute the largest value in a vector, 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 a vector, 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 – 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 a vector, 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):

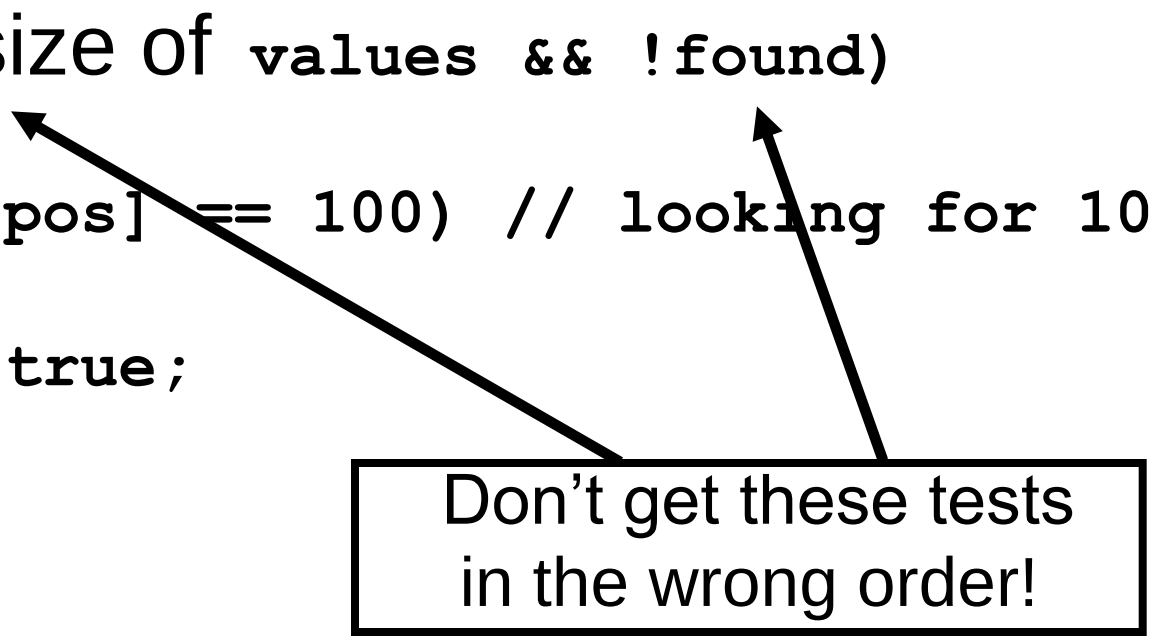
```
1 | 4 | 9 | 16 | 25
```

```
for (int i = 0; i < size of values; i++)
{
   if (i > 0)
   {
      cout << " | ";
   }
   cout << 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) // looking for 100
   {
      found = true;
   }
   else
   {
      pos++;
   }
}
```

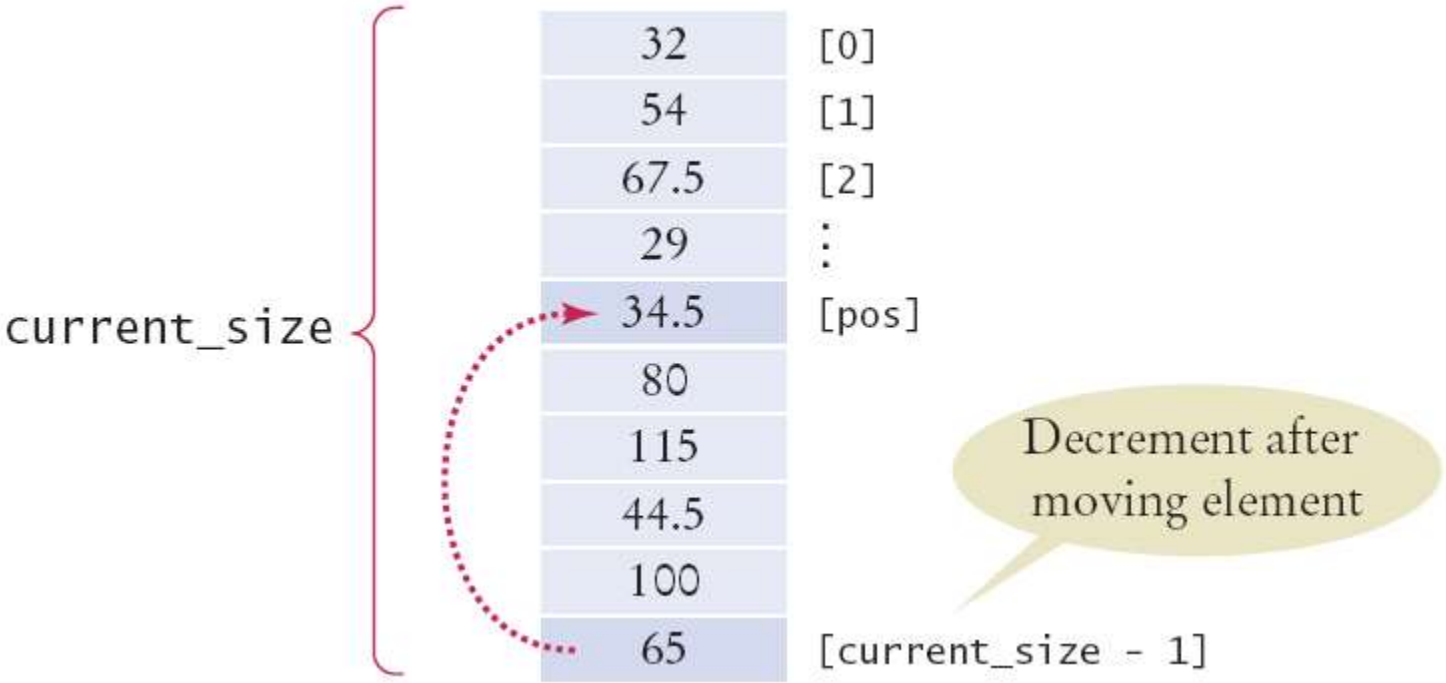


Common Algorithms – Removing an Element, Unordered

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

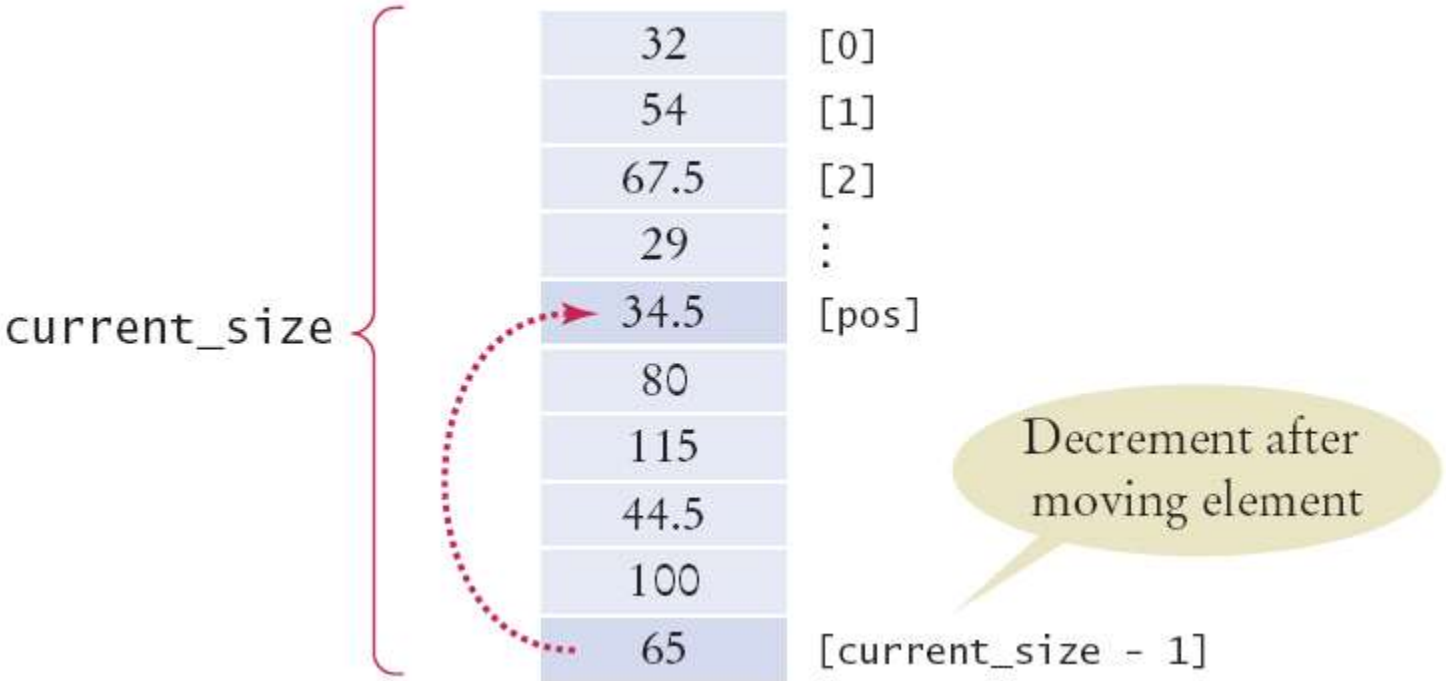
If the elements in the vector are not in any particular order, that task is easy to accomplish.

Simply overwrite the element to be removed with the *last* element of the vector, then shrink the size of the vector by removing the value that was copied.



Common Algorithms – Removing an Element, Unordered

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



Common Algorithms – Removing an Element, Ordered

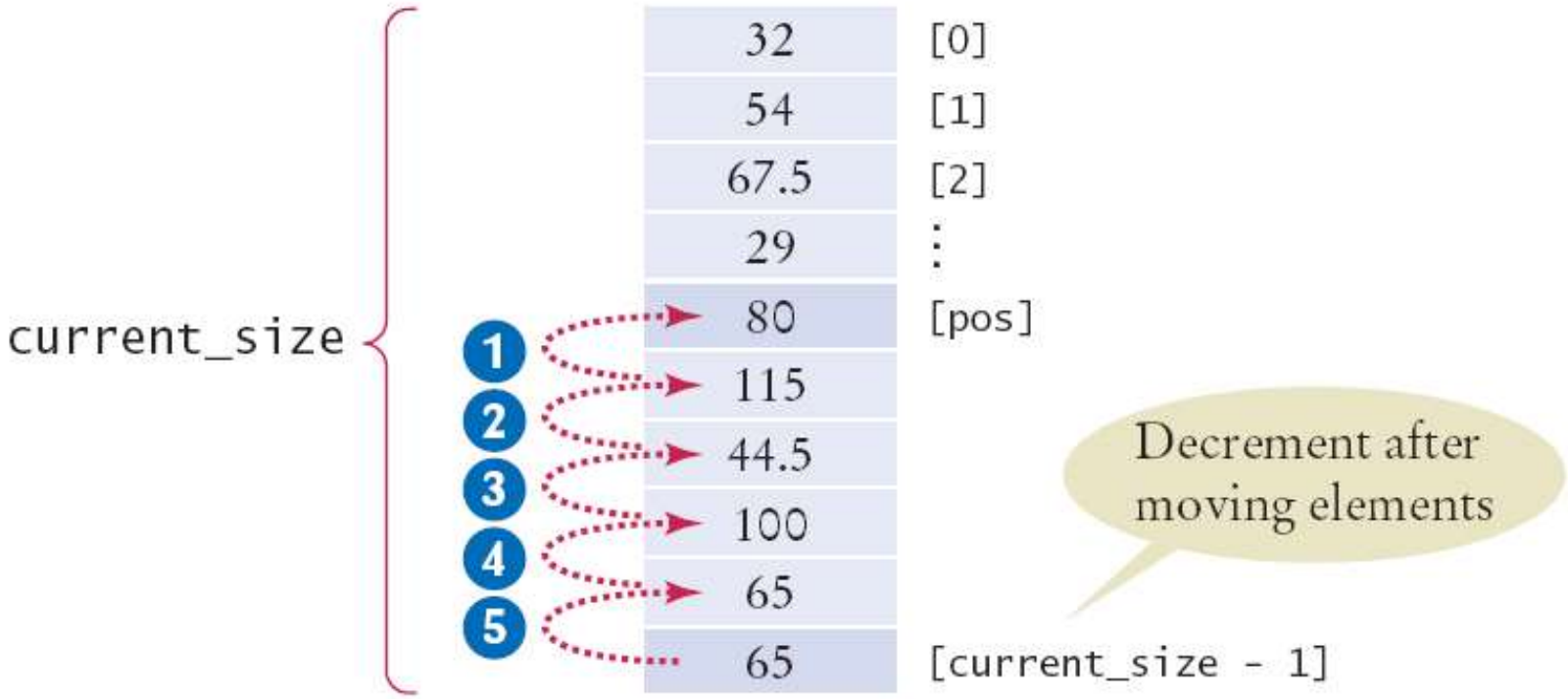
The situation is more complex
if the order of the elements matters.

Then you must move all elements following the element to be removed “down” (to a lower index), and then 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

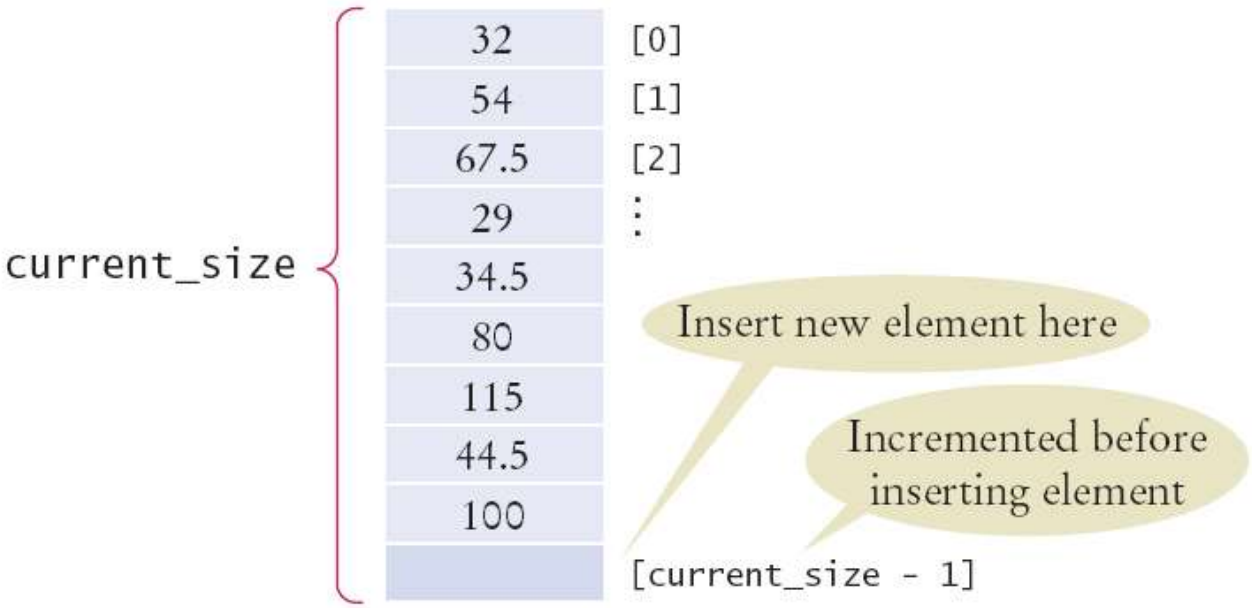
```
for (int i = pos + 1; i < current_size; i++)
{
   values[i - 1] = values[i];
}
current_size--;
```



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;
}
```

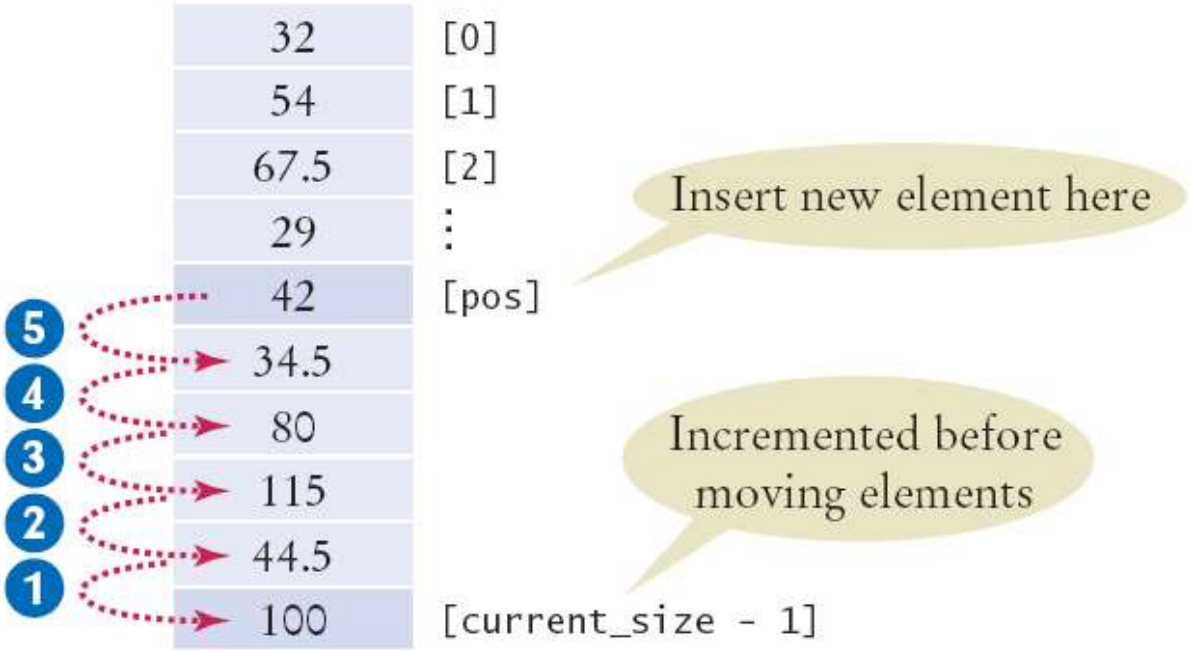


Common Algorithms – Inserting an Element Ordered

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 vector must be moved "up".

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



Common Algorithms – Inserting an Element Ordered

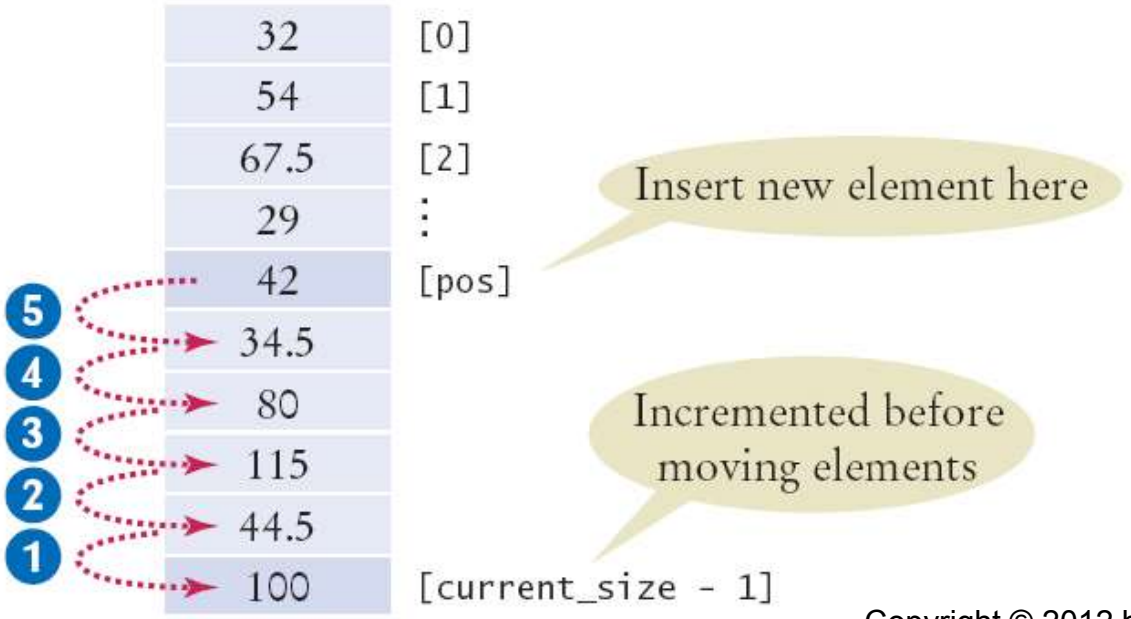
First, you must make the array one larger by incrementing **`current_size`**.

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

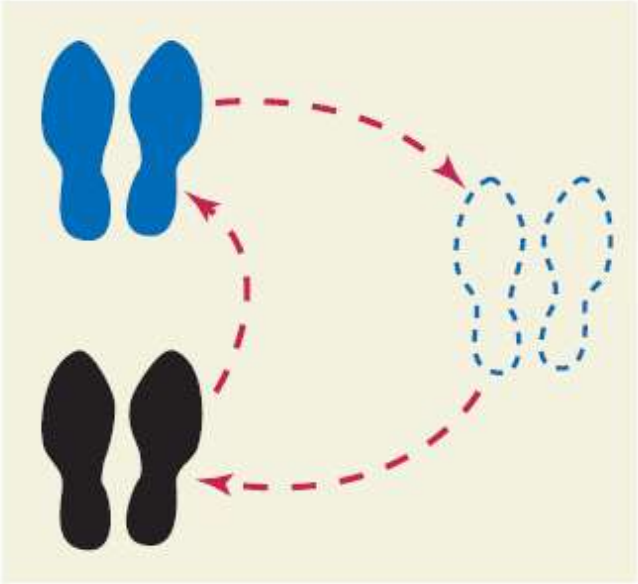
```
if (current_size < CAPACITY)
{
   current_size++;
   for (int i = current_size - 1; i > pos; i--)
   {
      values[i] = values[i - 1];
   }
   values[pos] = new_element;
}
```



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`.

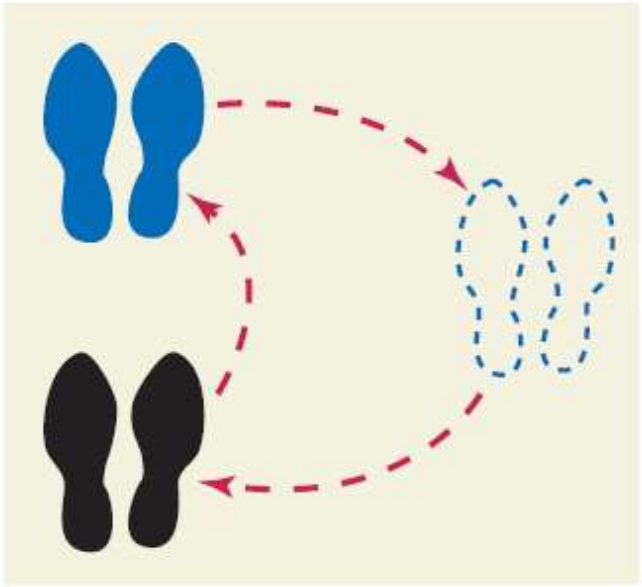
Then what?

Put' `j`'s value back in `j` in the second line?

ARGHHH!

Common Algorithms – Swapping Elements

You need a *third* dance partner!

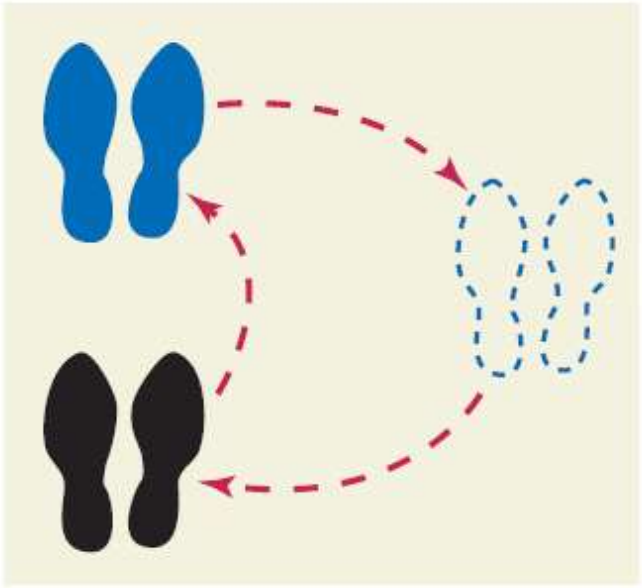


Common Algorithms – Swapping Elements

Let's Waltz!

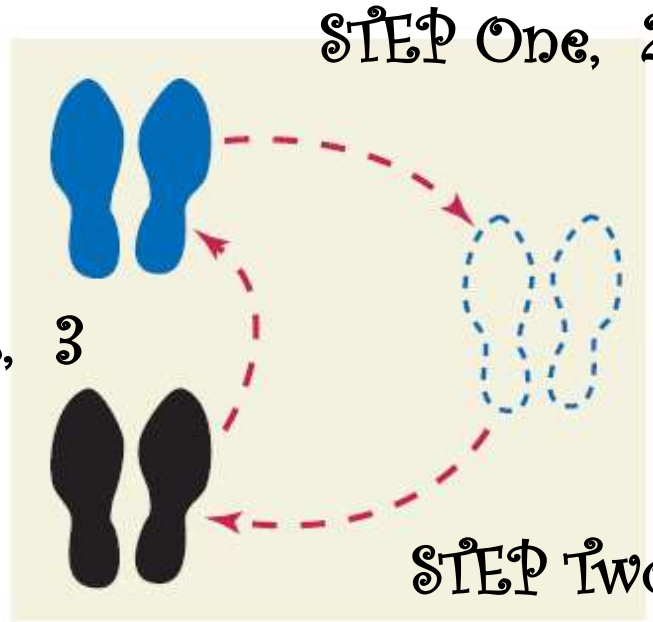
1, 2, 3,

go, 2, 3,



Common Algorithms – Swapping Elements

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



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++)
{
   cin >> values[i];
}
```

Common Algorithms – Reading Input

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

But not hopeless.

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;
while (cin >> input)
{
   if (current_size < CAPACITY)
   {
      values[current_size] = input;
      current_size++;
   }
}
```

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

ch06/largest.cpp

```
#include <iostream>

using namespace std;

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

Common Algorithms – Maximum

ch06/largest.cpp

```
cout << "Please enter values, Q to quit:" << endl;
double input;
while (cin >> input)
{
   if (current_size < CAPACITY)
   {
      values[current_size] = input;
      current_size++;
   }
}
```

Common Algorithms – Maximum

ch06/largest.cpp

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

Common Algorithms – Maximum

ch06/largest.cpp

```
   for (int i = 0; i < current_size; i++)
   {
      cout << values[i];
      if (values[i] == largest)
      {
         cout << " <== largest value";
      }
      cout << endl;
   }

   return 0;
}
```

Sorting with the C++ Library

Getting data into order is something that is often needed.

An alphabetical listing.

A list of grades in descending order.

Sorting with the C++ Library

In C++, you call the **`sort`** function
to do your sorting for you.

But the syntax is new to you:

Recall our **`values`** array
with the companion variable **`current_size`**.

```
sort(values, values + current_size);
```

To sort the elements into ascending numerical order,
you call the **`sort`** algorithm:

Sorting with the C++ Library

You will need to write:

```
#include <algorithm>
```

in order to use the **`sort`** function.

```
sort(values, values + current_size);
```

Sorting with the C++ Library

Notice also that you must tell the **`sort`** function where to begin: **`values`**,
(which is the start of the array)
and where to end: **`values + current_size`**,
(which is one *after* the last element in the array).

```
sort(values, values + current_size);
```

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.

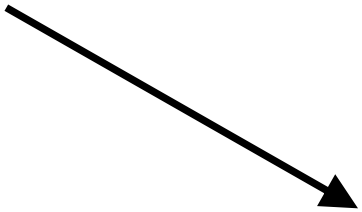
Arrays as Parameters in Functions

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

No, that is not a box!



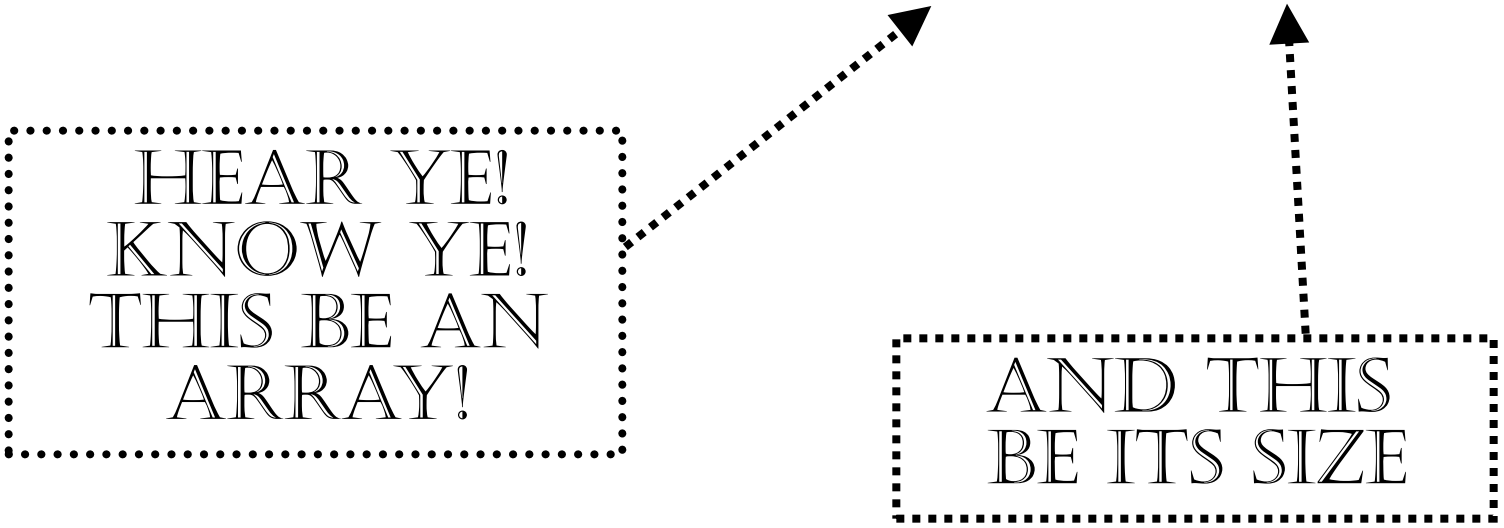
```
double sum(double data[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++)
   {
      total = total + data[i];
   }
   return total;
}
```

It is an empty pair of square brackets.

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.

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

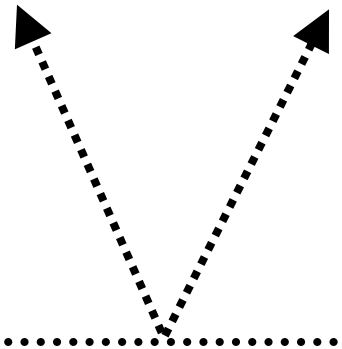


Arrays as Parameters in Functions

NE'ER ERR!

FAIL YE NOT TO

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



PROFFER BOTH - THUSLY!

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);
```

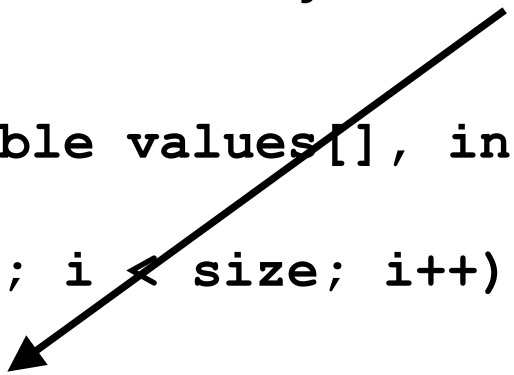
You can also pass a smaller size to the function:

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

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

Arrays as Parameters in Functions

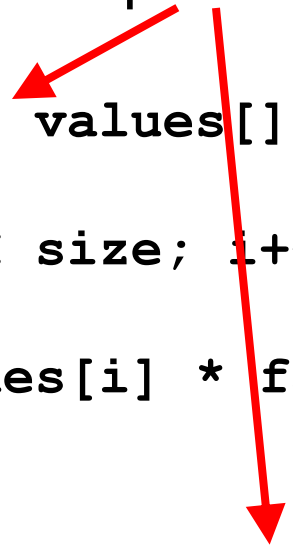
When you pass an array into a function,
the contents of the array can ***always*** be changed:



```
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

And writing an ampersand is *always* *an error*:



```
void multiply1(double& values[], int size, double factor)
{
   for (int i = 0; i < size; i++)
   {
      values[i] = values[i] * factor;
   }
}
void multiply2(double values[]&, int size, double factor)
{
   for (int i = 0; i < size; i++)
   {
      values[i] = values[i] * factor;
   }
}
```

Arrays as Parameters in Functions

And writing an ampersand is *always* an error:

```
void multiply1(double& values[], int size, double factor)
{
   for (int i = 0; i < size; i++)
   {
      values[i] = values[i] * factor;
   }
}
```

```
void multiply2(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.

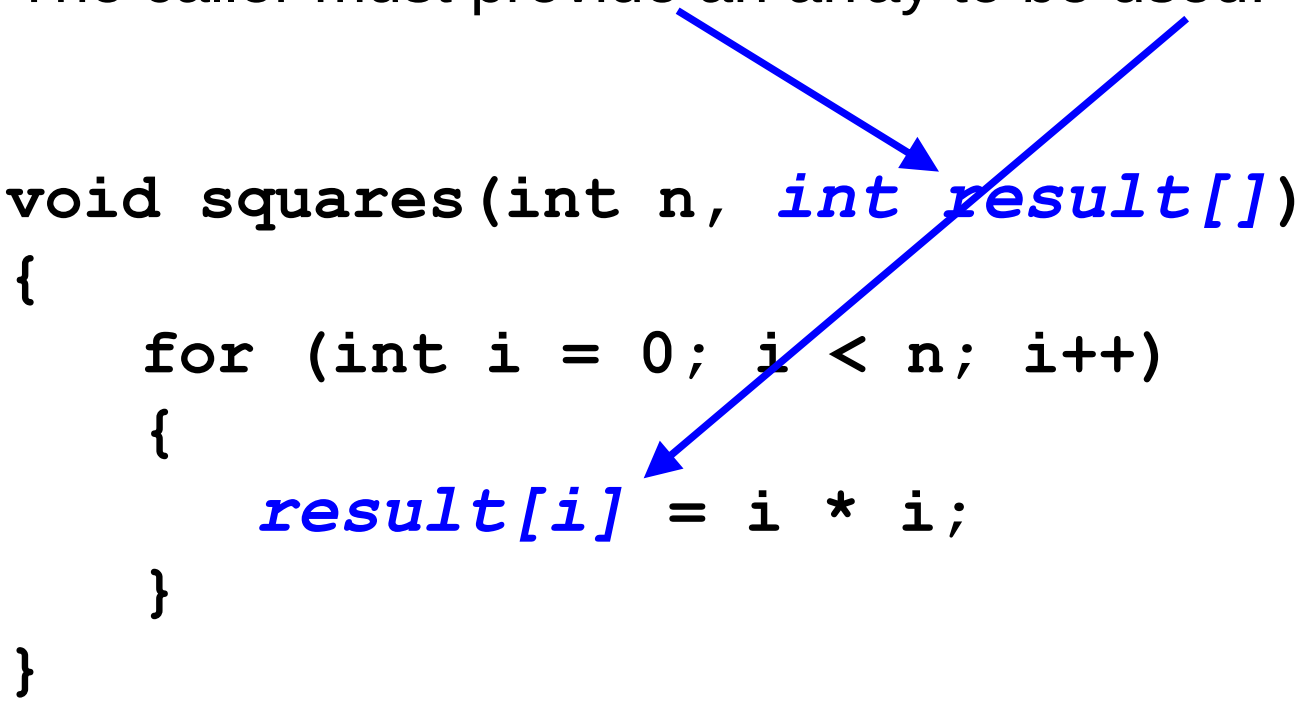
Arrays as Parameters in Functions

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

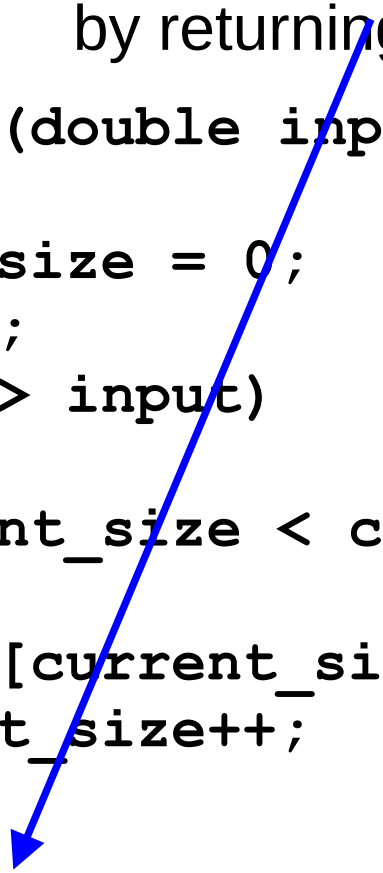
The caller must provide an array to be used:



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

Arrays as Parameters in Functions

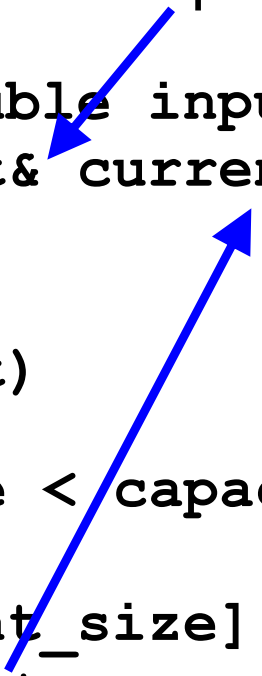
Here is a call to the function:

```
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

Or it can let the caller know by using a reference parameter:



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

Arrays as Parameters in Functions

Here is a call to the reference parameter version of **`append_inputs`**:

```
append_inputs(values, MAXIMUM_NUMBER_OF_VALUES,
              current_size);
```

As before, after the call, the **`current_size`** variable specifies how many are in the array.

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 <iostream>
using namespace std;
```

ch06/functions.cpp

```
/**
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)
{
   int current_size = 0;
   cout << "Please enter values, Q to quit:" << endl;
   bool more = true;
   while (more)
   {
```

Arrays as Parameters in Functions

ch06/functions.cpp

```
      double input;
      cin >> input;
      if (cin.fail())
      {
         more = false;
      }
      else if (current_size < capacity)
      {
         inputs[current_size] = input;
         current_size++;
      }
   }
   return current_size;
}
```

Arrays as Parameters in Functions

ch06/functions.cpp

```
/**
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

ch06/functions.cpp

```
/**
Prints the elements of a vector, 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) { cout << ", "; }
      cout << values[i];
   }
   cout << endl;
}
```

Arrays as Parameters in Functions

ch06/functions.cpp

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

   return 0;
}
```

End Arrays and Vectors I

Slides by Evan Gallagher & Nikolay Kirov