BLG 102E
Introduction to Scientific Computing and Engineering

SPRING 2025

WEEK 7

ISTANBUL TECHNICAL UNIVERSITY

Arrays

Slides credit: Evan Gallagher

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)

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;
```

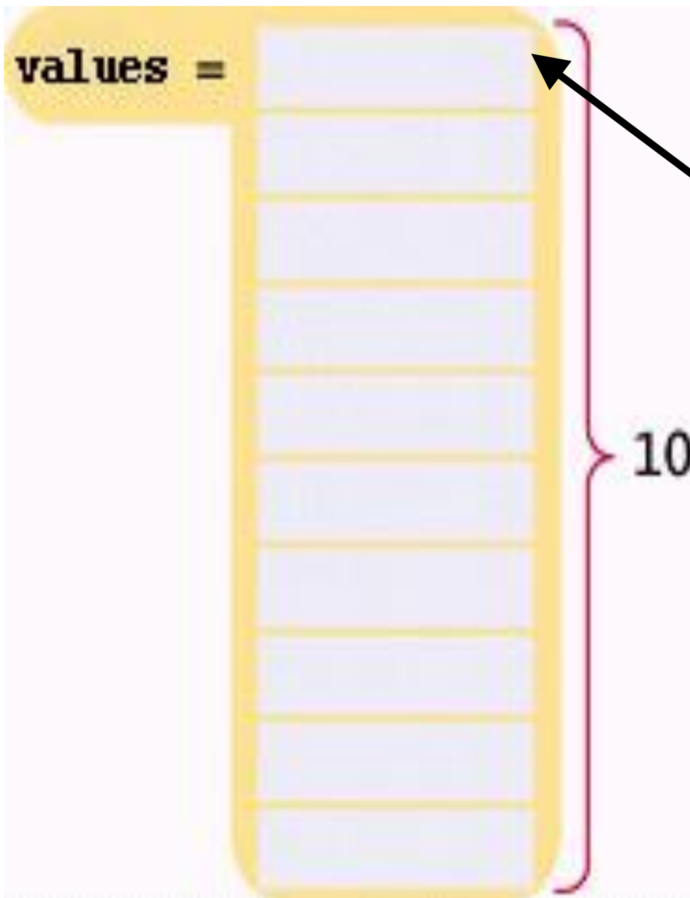
Then what???

Using Arrays

- Arrays
  - Structures of related data items
  - Static entity
    - same size throughout program
- Array definition
  - Group of consecutive memory locations
  - Same name and data type

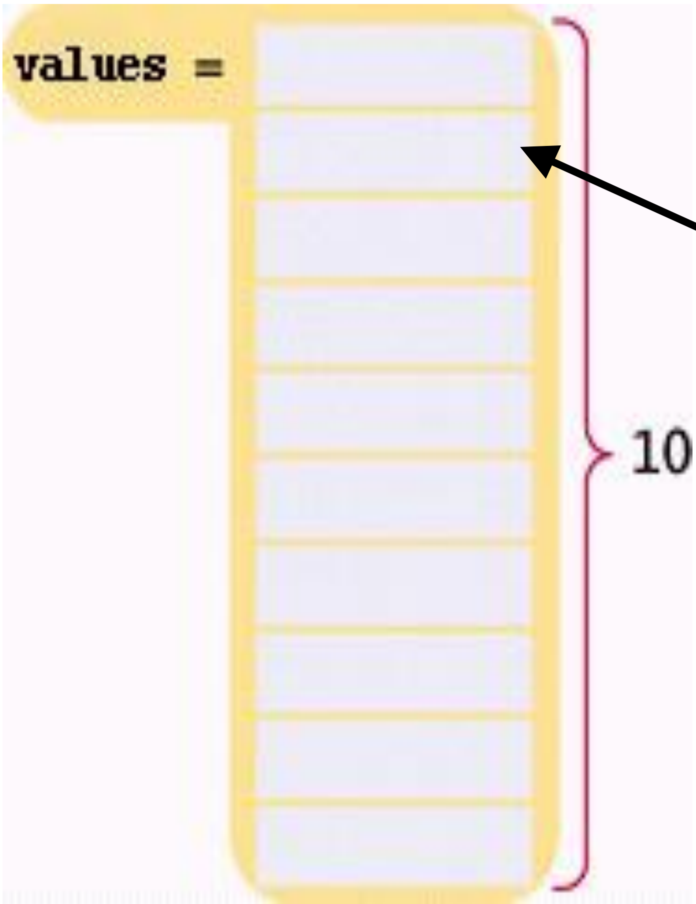
Using Arrays

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



Hm. Is this the max, so far?

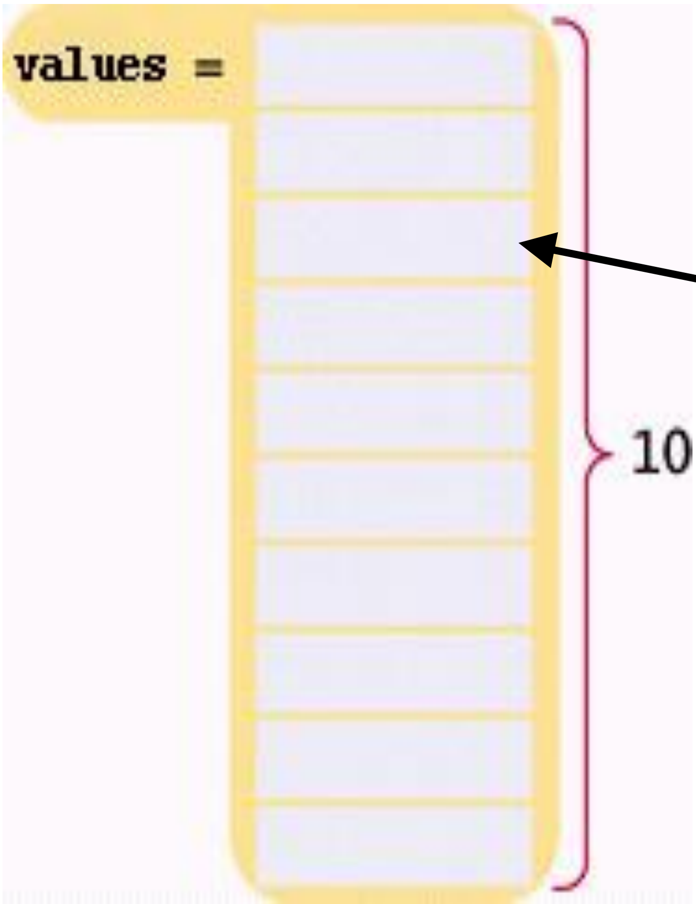
Using Arrays



Maybe this one?

©

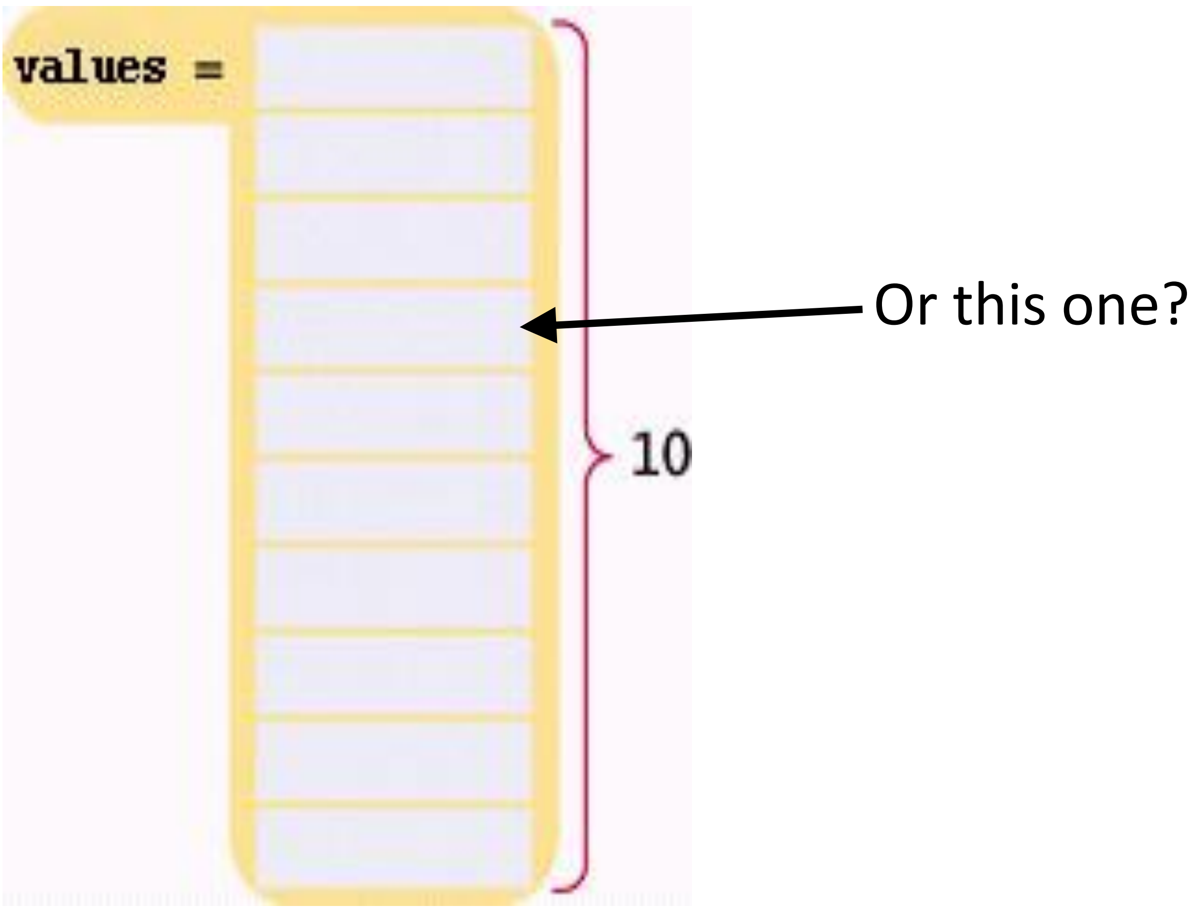
Using Arrays



Or this one?

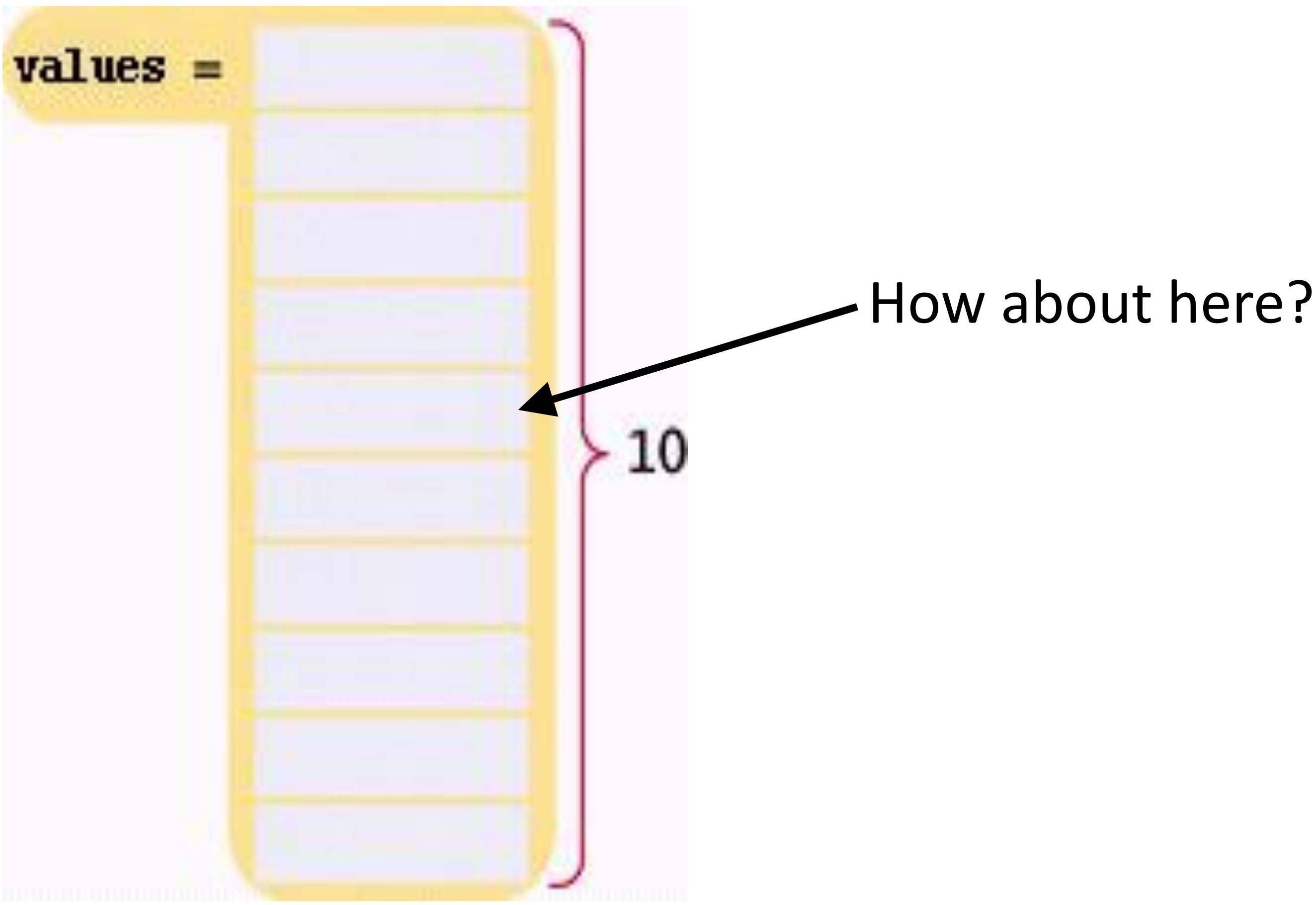
©

Using Arrays

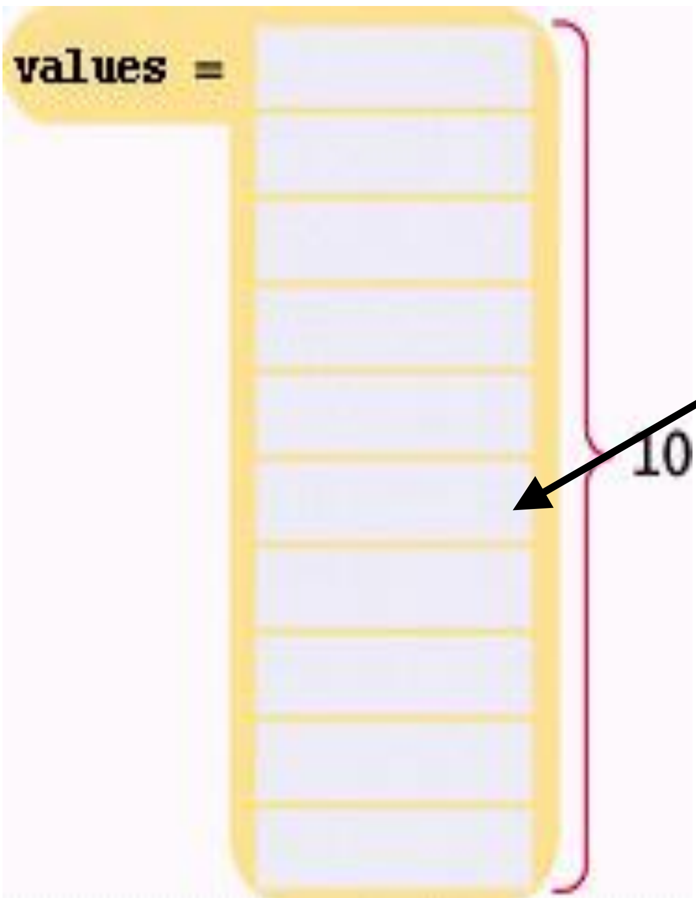


©

Using Arrays

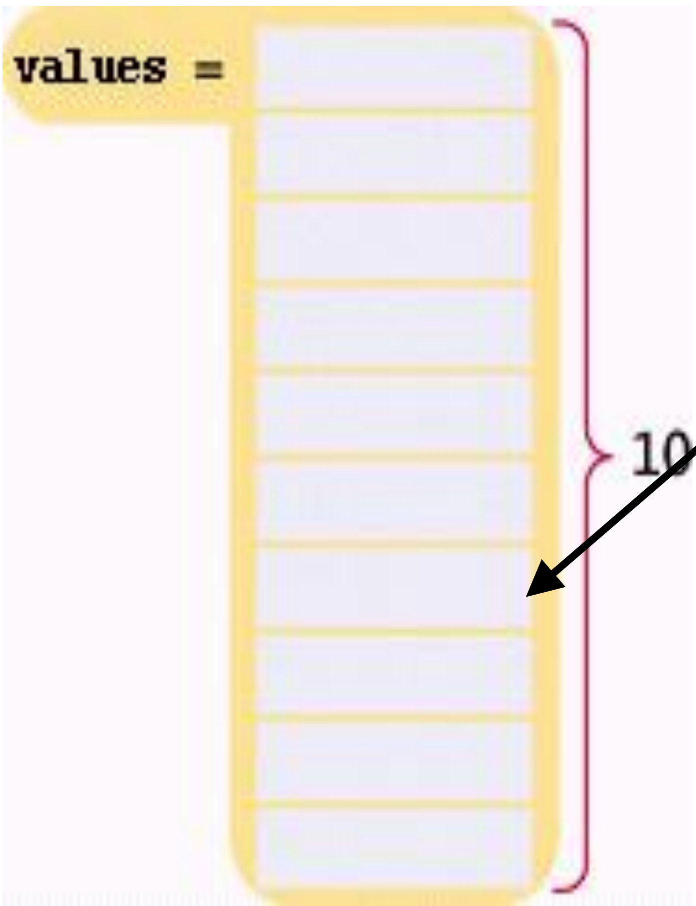


Using Arrays



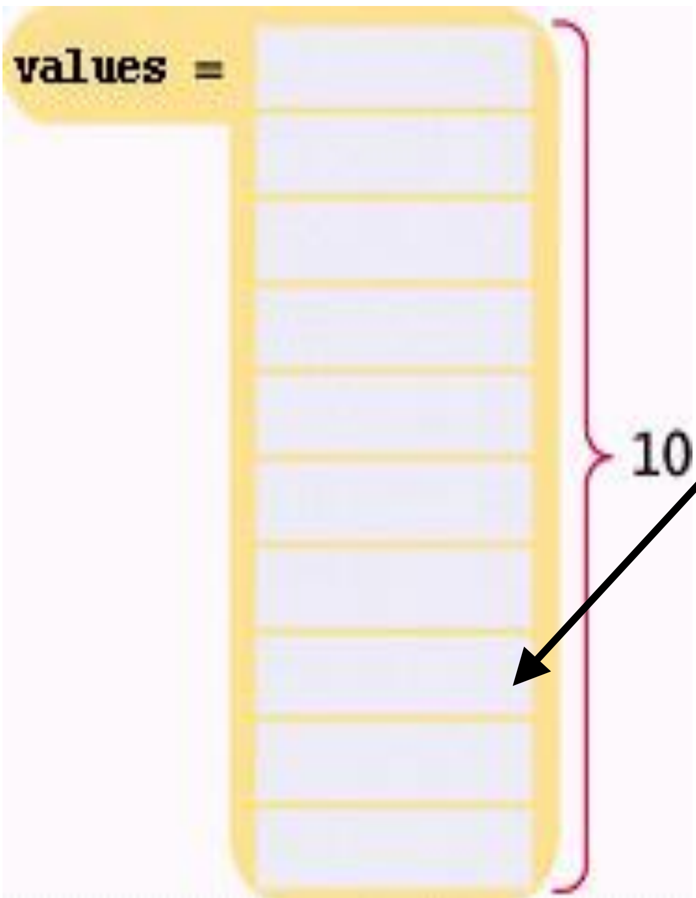
Gotta check here too!

Using Arrays



Again, maybe this one?

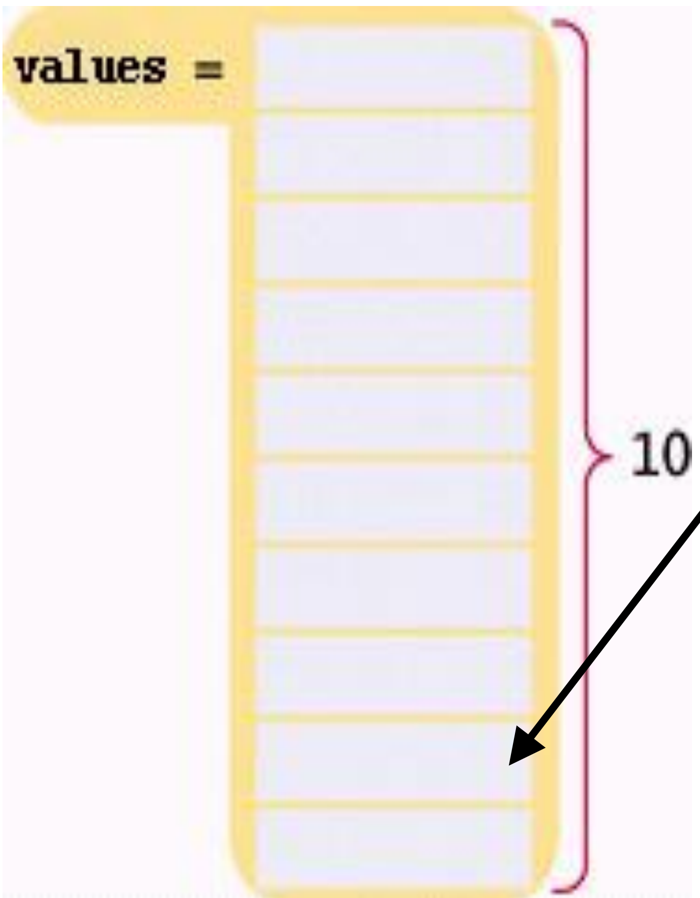
Using Arrays



Or this one?

©

Using Arrays

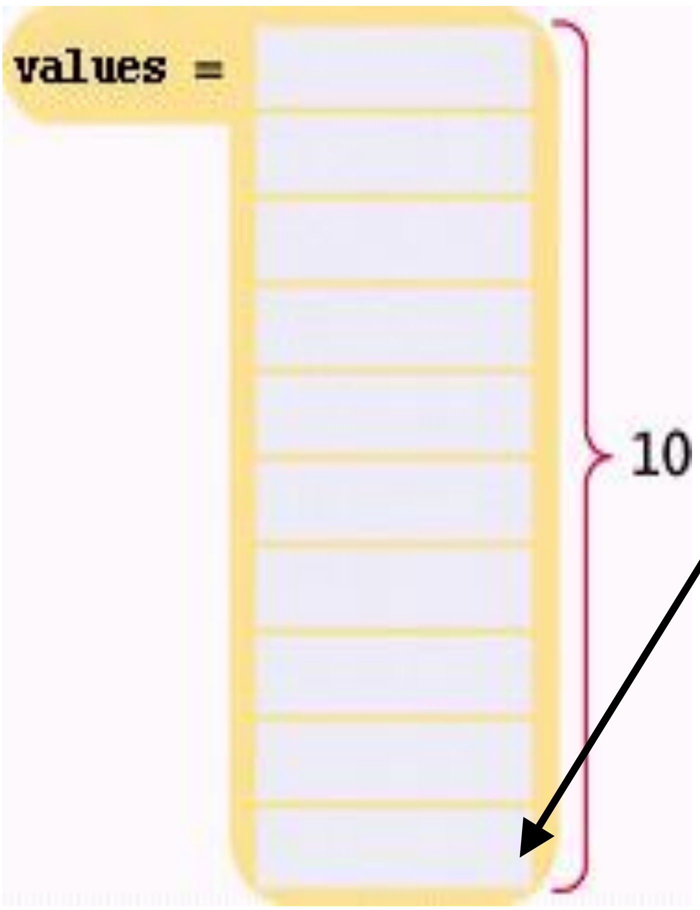


Or this one?

Will this never end!

©

Using Arrays



Or the last one? *Finally!*

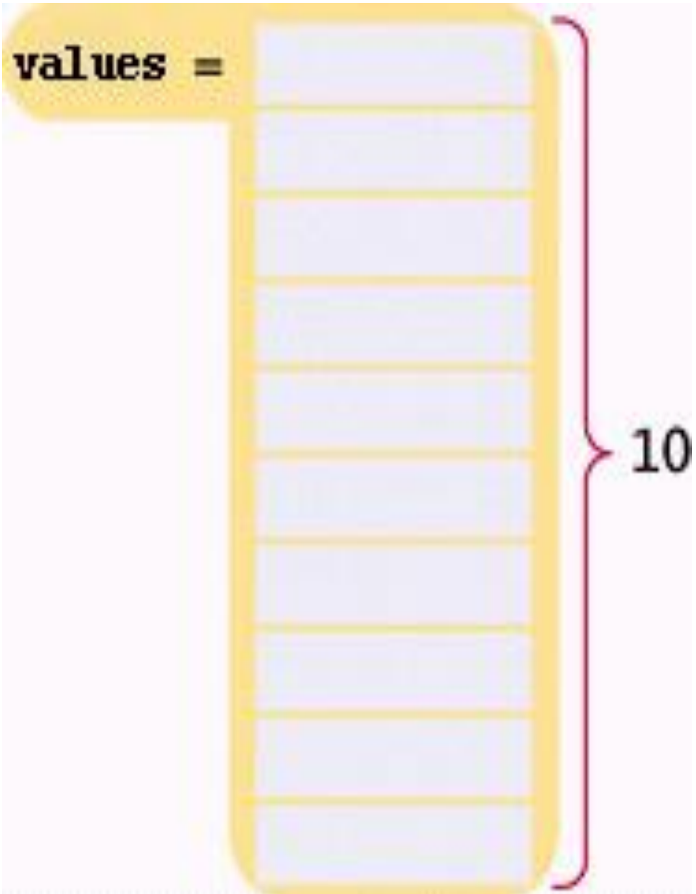
Using Arrays

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 as an array

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

type of each element

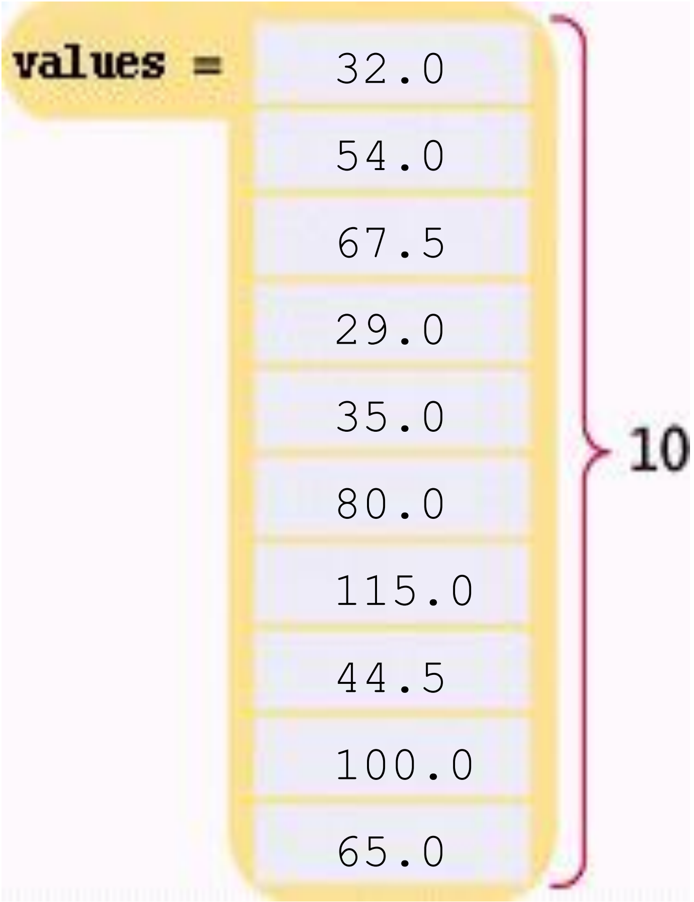
quantity of elements – the “size” of the array, must be a constant

©

Defining Arrays with Initialization

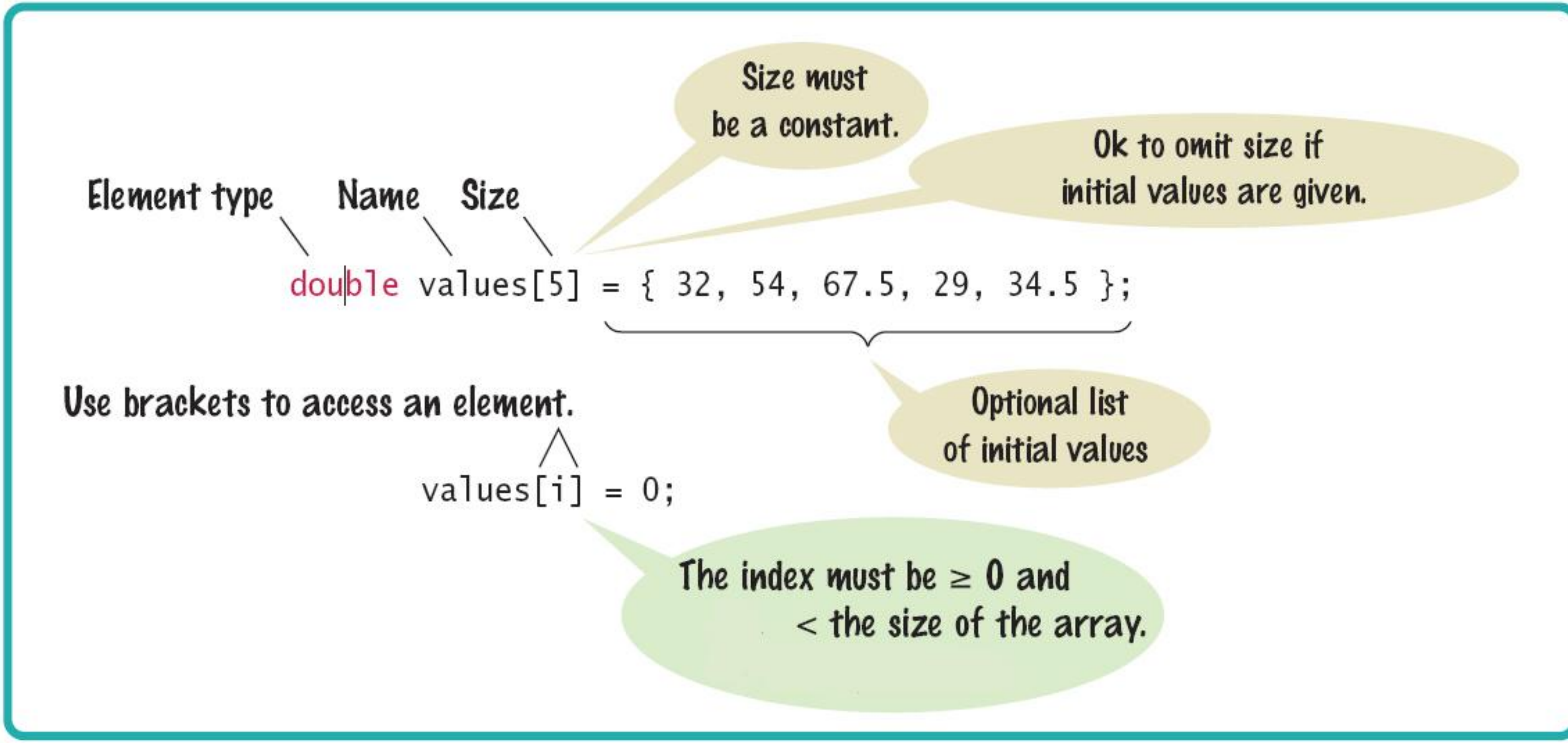
When you define an array, you can specify the initial values:

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



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`.

Initializing Array Elements

- We can use a list of initializers in declaration statement.

```
int a[ 5 ] = { 10, 20, 30, 40, 50 };
```

  - If not enough initializers, rightmost elements become 0

```
int a[ 5 ] = { 0 };   // All elements are set to zero
int a[ 5 ] = { 10 };   // First element is 10, other elements are zero
```

  - If too many elements, then a syntax error is produced
  - C arrays have no bounds checking
    Example: a[200] = 60; The statement will not give compiler error.

- If size omitted, count of initializers will determine the size

```
int a[ ] = { 10, 20, 30, 40, 50 };
```

  - 5 initializers, therefore compiler knows that array has 5 elements

Example: Initializing an array with a declaration

- Program defines and initializes an array.

```c
/* Initializing an array with a initializer list */
#include <stdio.h>

int main()
{
   // use initializer list to initialize array n
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
   int i; // counter

   printf("Element  Value \n" );

   // output contents of array in tabular format
   for ( i = 0; i < 10; i++ ) {
      printf( "%7d %13d \n", i, n[ i ] );
   }

} // end main
```

Program
Output

```
Element        Value
      0           32
      1           27
      2           64
      3           18
      4           95
      5           14
      6           90
      7           70
      8           60
      9           37
```

Accessing an Array Element

An array element can be used like any variable.

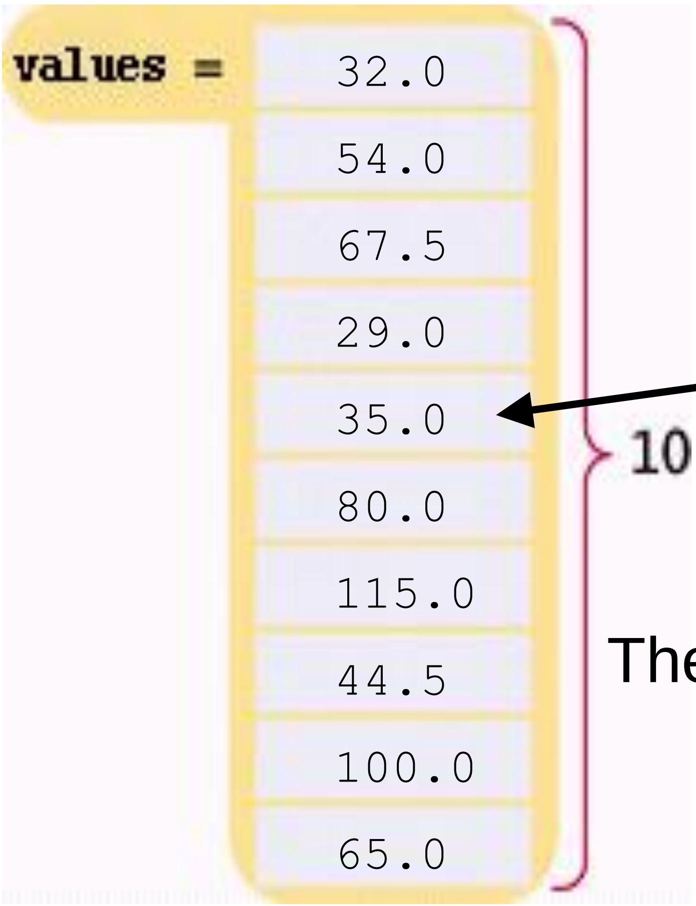
To access an array element, you use the notation:

```
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*.

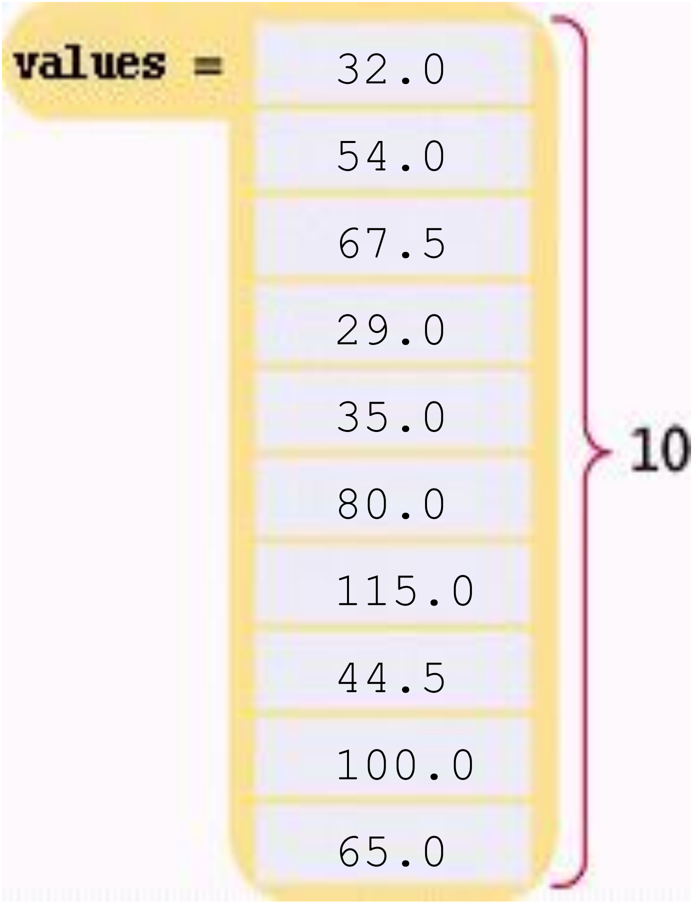



```
double values[10];
...
printf("%f\n", values[4]);
```

The output will be `35.0`.

Accessing an Array Element

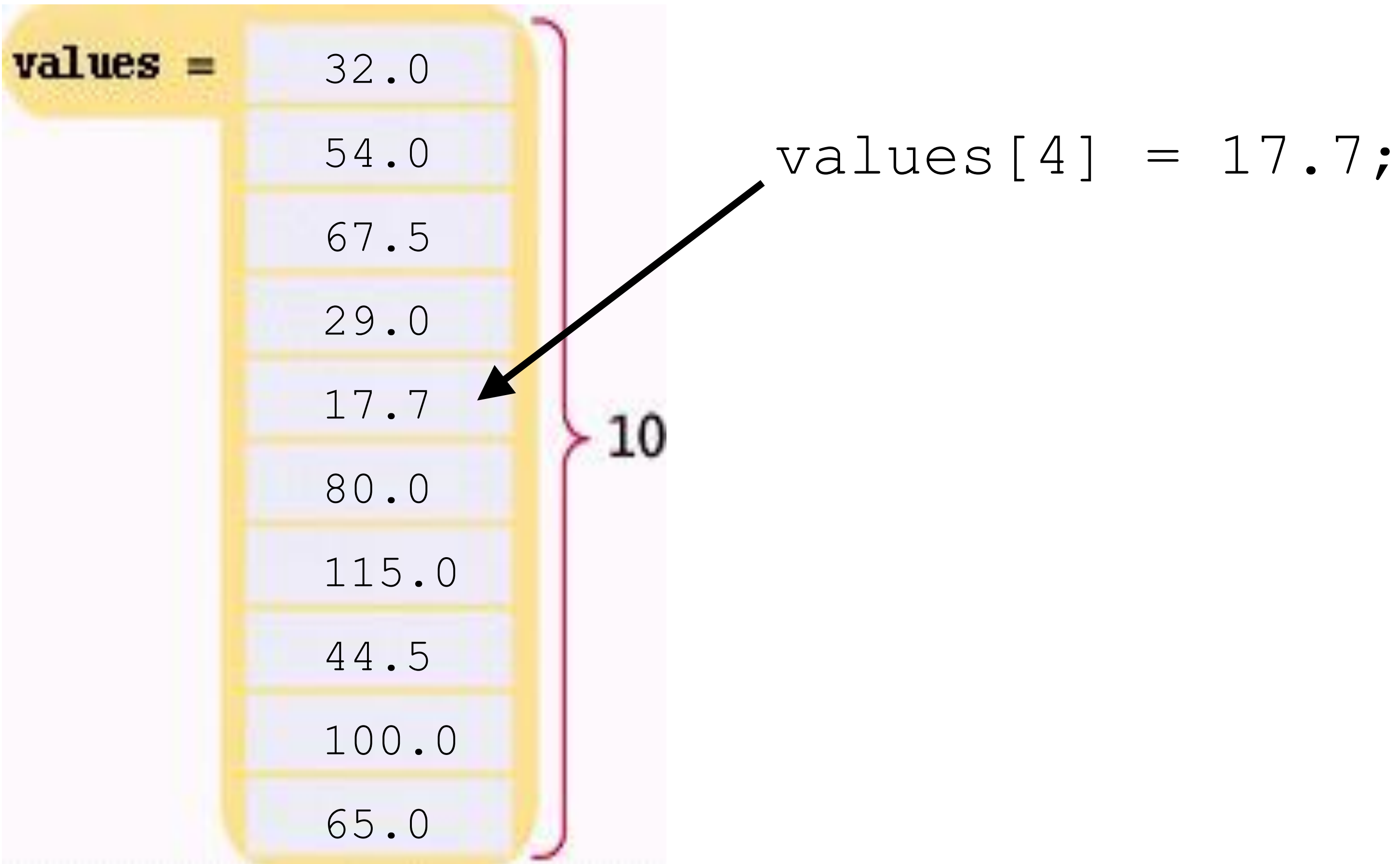
The same notation can be used to change the element.




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

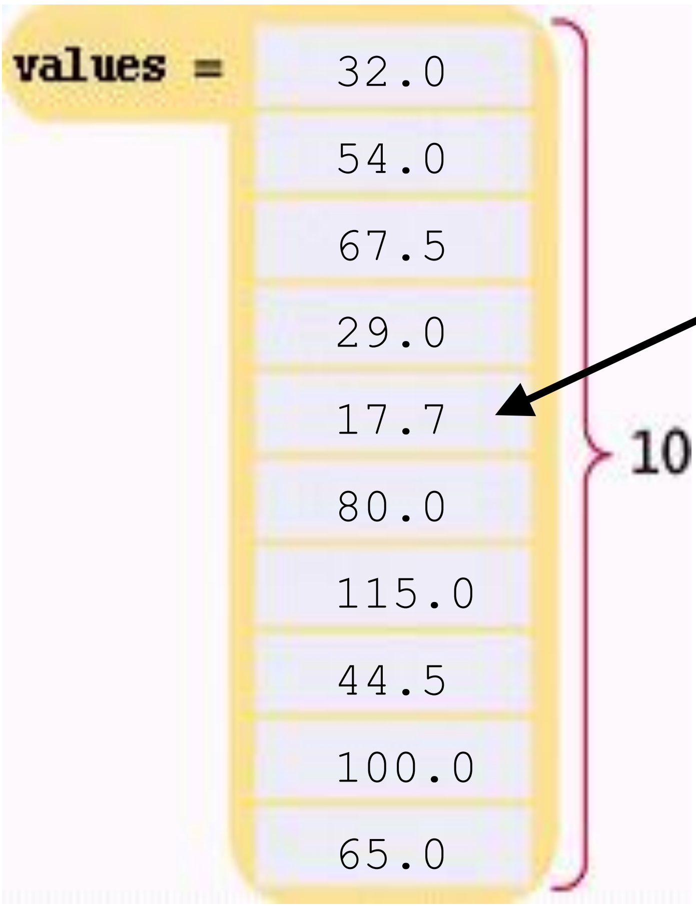
Accessing an Array Element

The same notation can be used to change the element.



Accessing an Array Element

The same notation can be used to change the element.



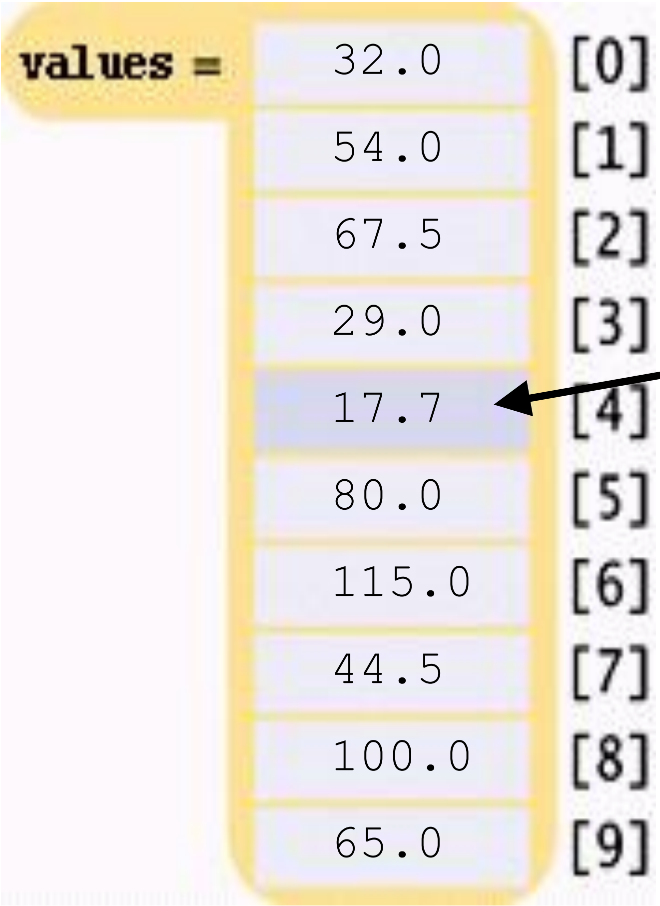

```
values[4] = 17.7;

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

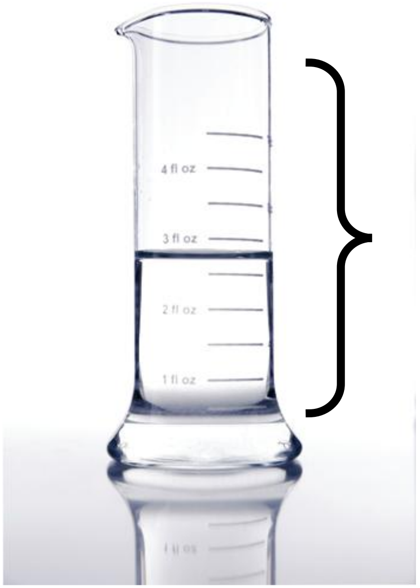
That is, the legal elements for the `values` array are:

`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:

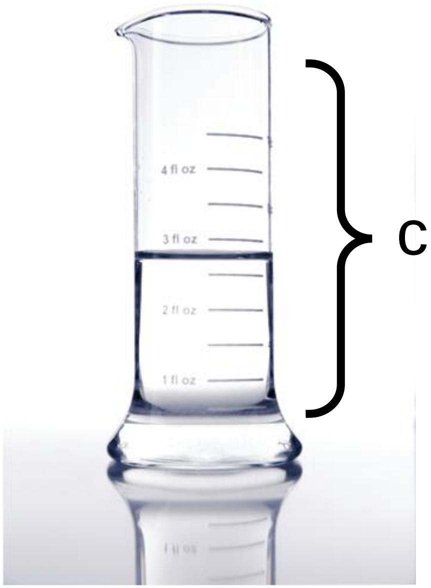


Does it always?
Just look at that beaker.
Guess not!

Partially-Filled Arrays – Capacity

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

We call this quantity the *capacity*.



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.

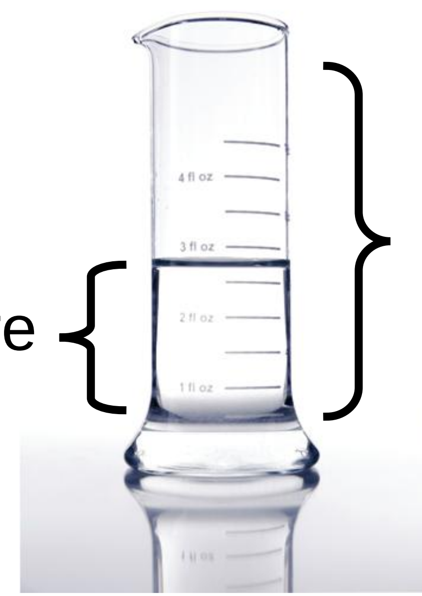
We would set the capacity with a `const`:

```
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:

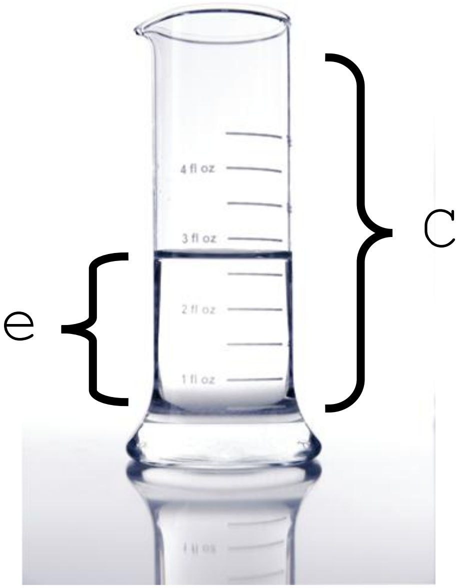
```
const int CAPACITY = 100;
double values[CAPACITY];

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
```

current_size

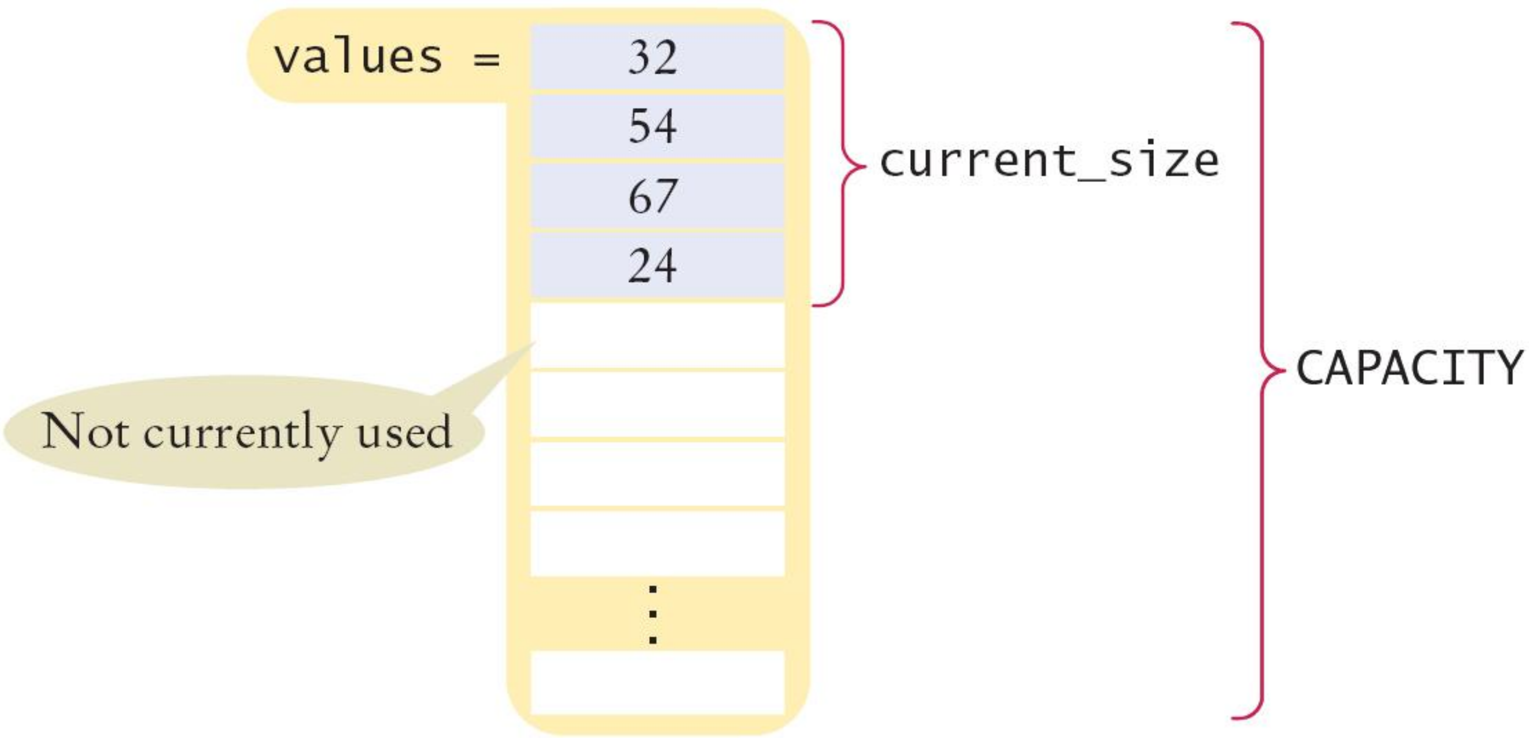


CAPACITY

©

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

Partially-Filled Arrays – Capacity

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

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

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

Doing this can corrupt data or cause your program to terminate.

DANGER!!!

DANGER!!!

DANGER!!!

©

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 `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 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!

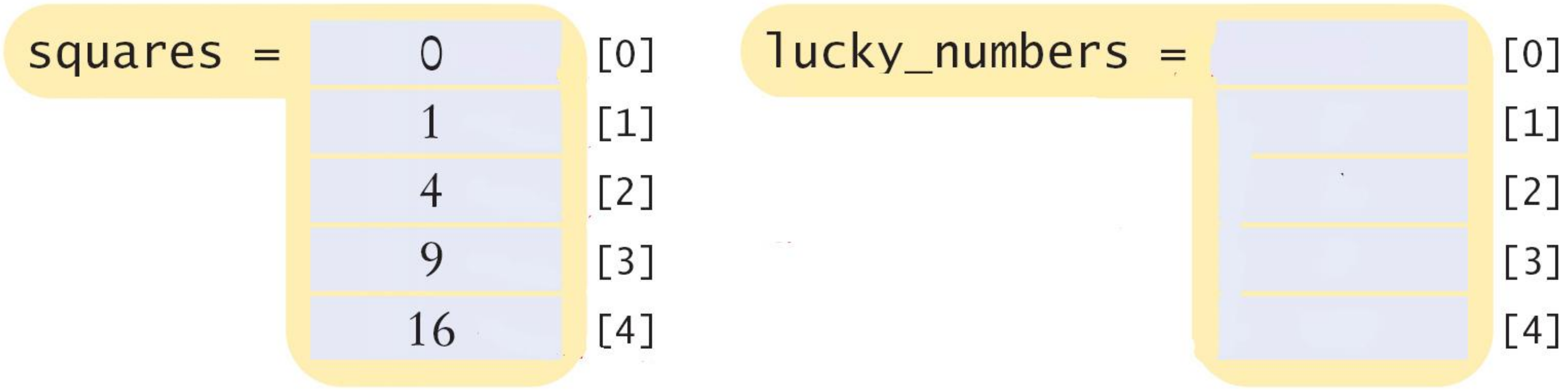
You cannot assign arrays!

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];
}
```

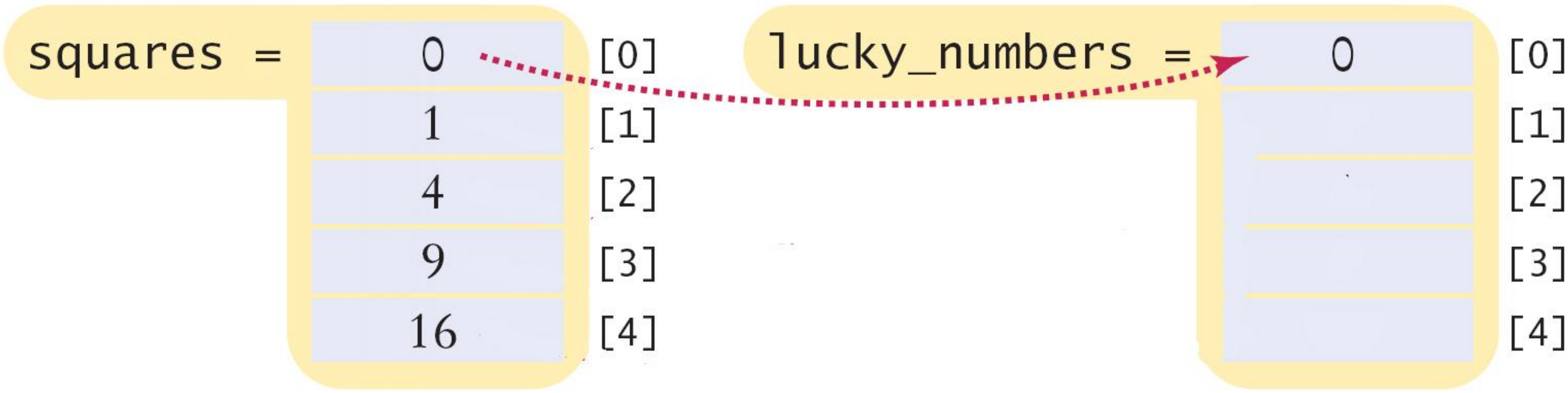
when `i` is `0`



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];
}
```

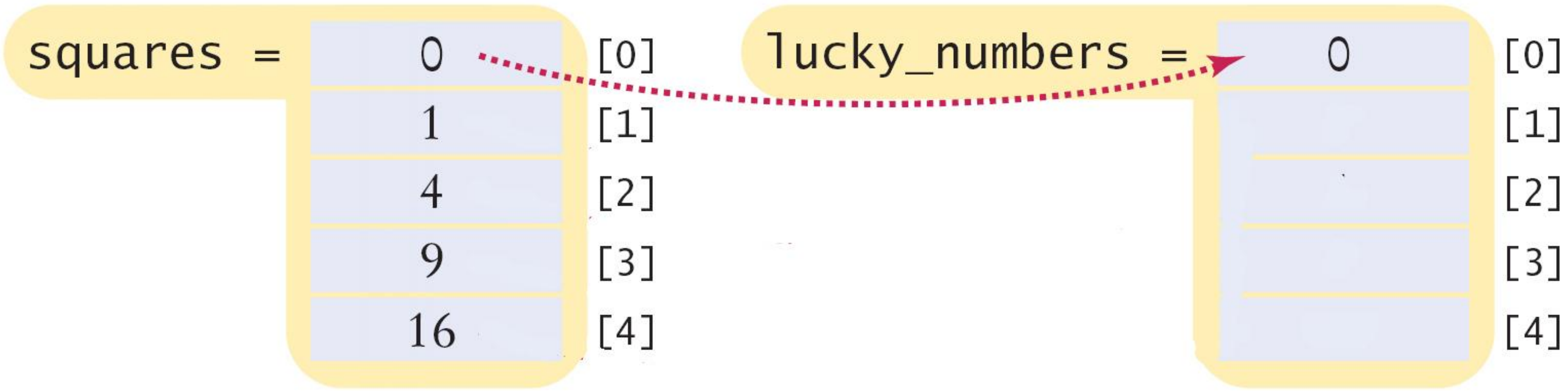
when `i` is `0`



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];
}
```

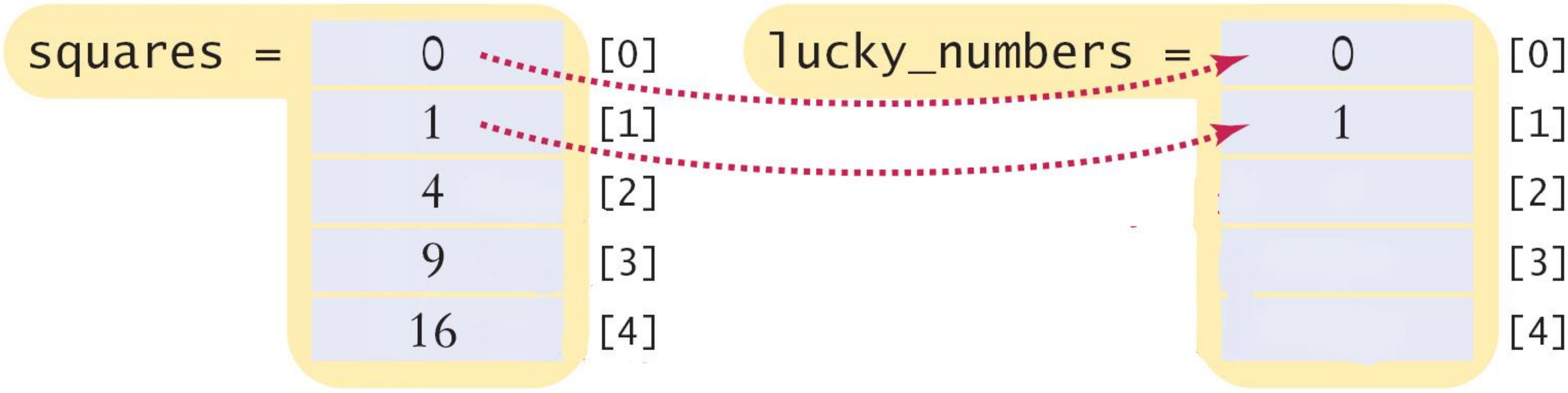
when `i` is `1`



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];
}
```

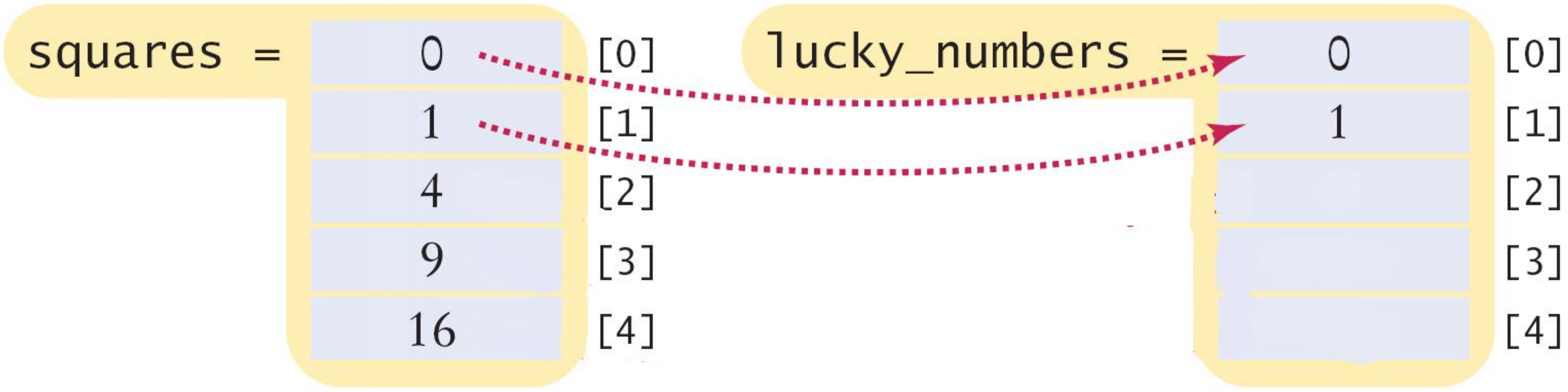
when `i` is `1`



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];
}
```

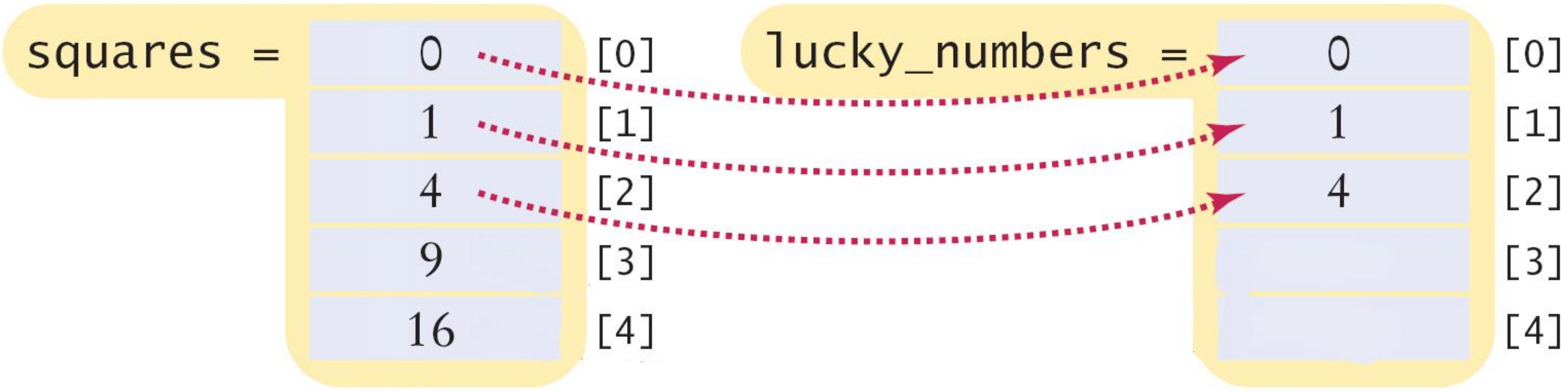
when `i` is `2`



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];
}
```

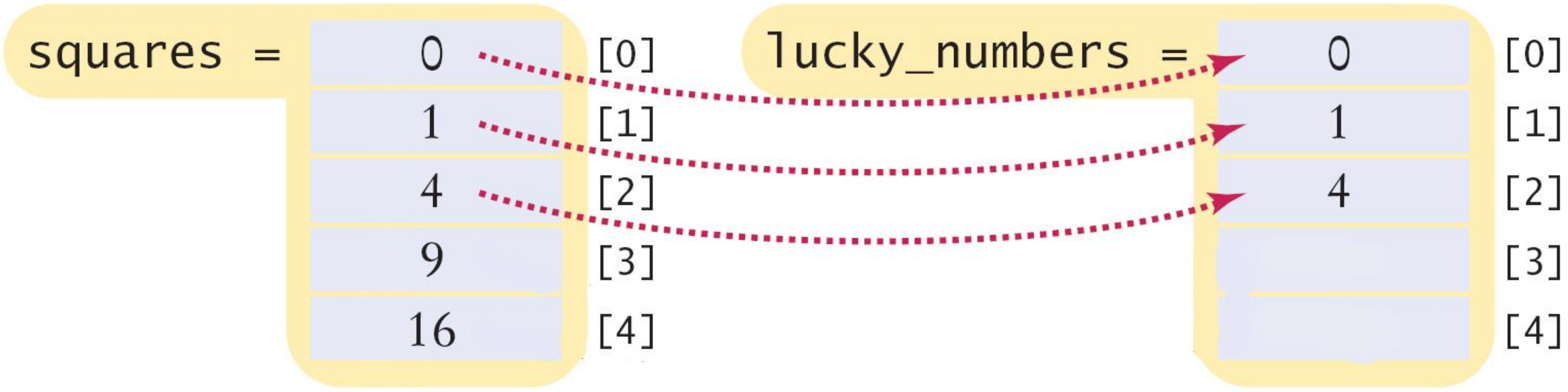
when `i` is `2`



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];
}
```

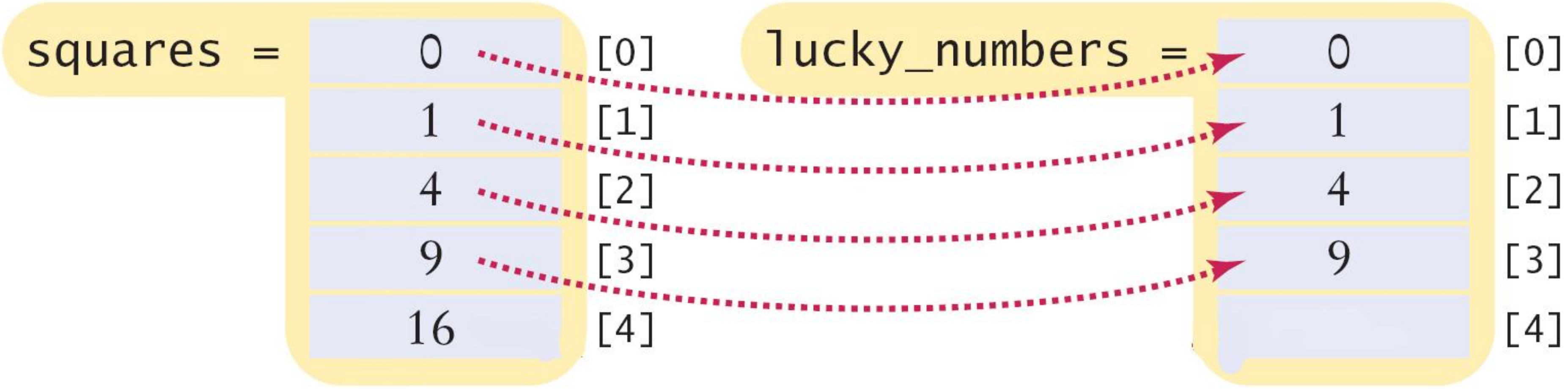
when `i` is `3`



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];
}
```

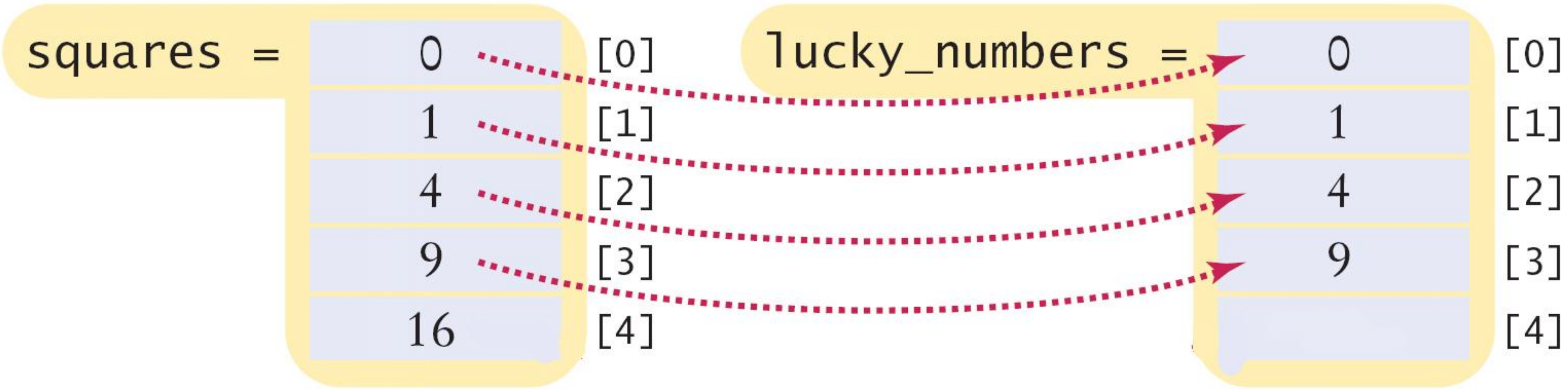
when `i` is `3`



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];
}
```

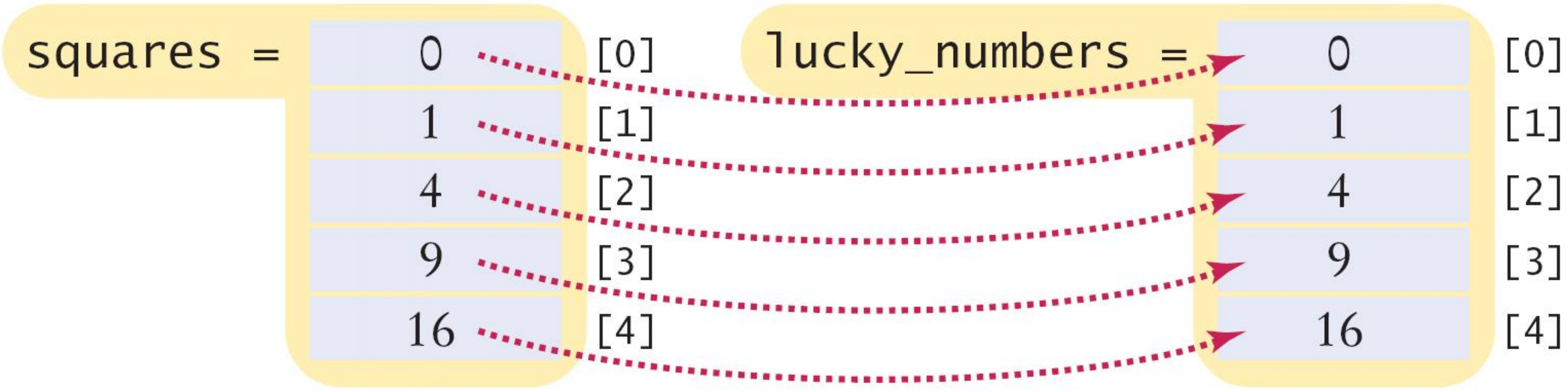
when `i` is `4`



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:

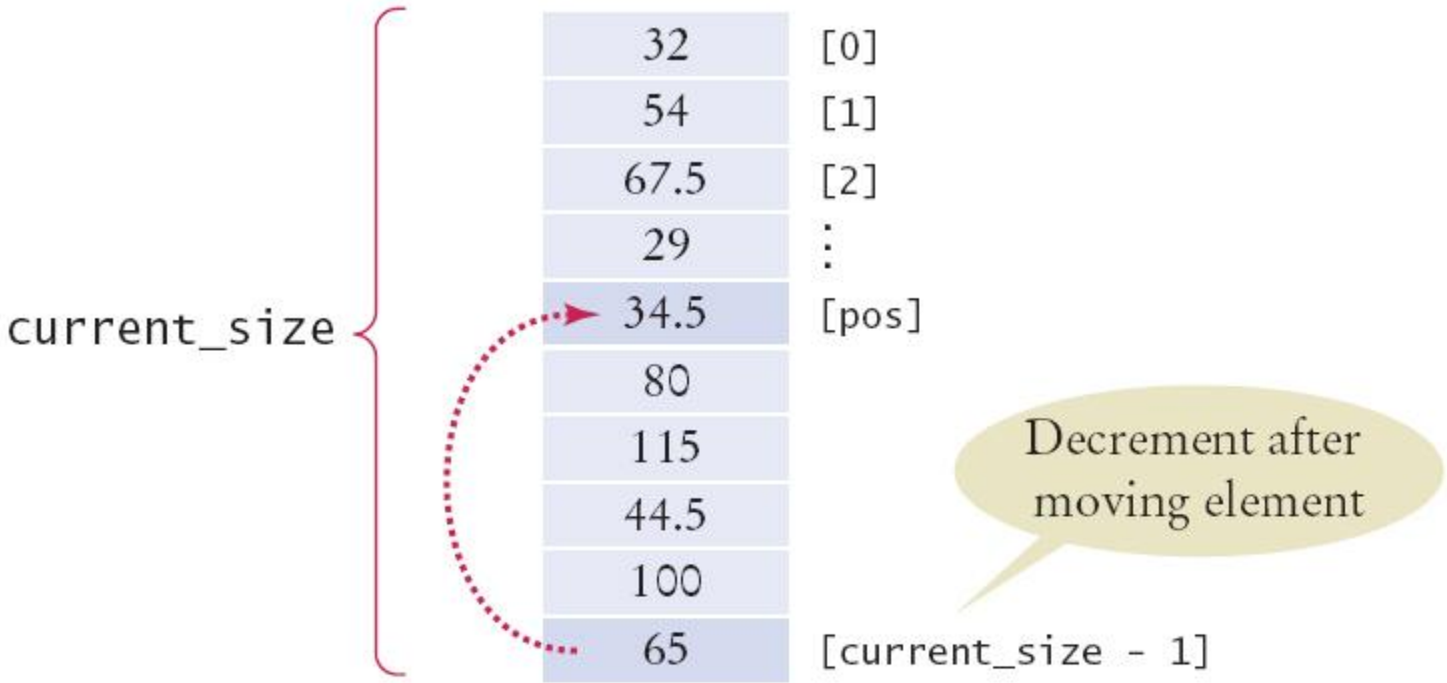
```
int pos = 0;
bool found = false;
while (pos < size of values && !found) {
    if (values[pos] == 100) {
        found = true;
    } else {
        pos++;
    }
}
```

Common Algorithms – Removing an Element, Unordered

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

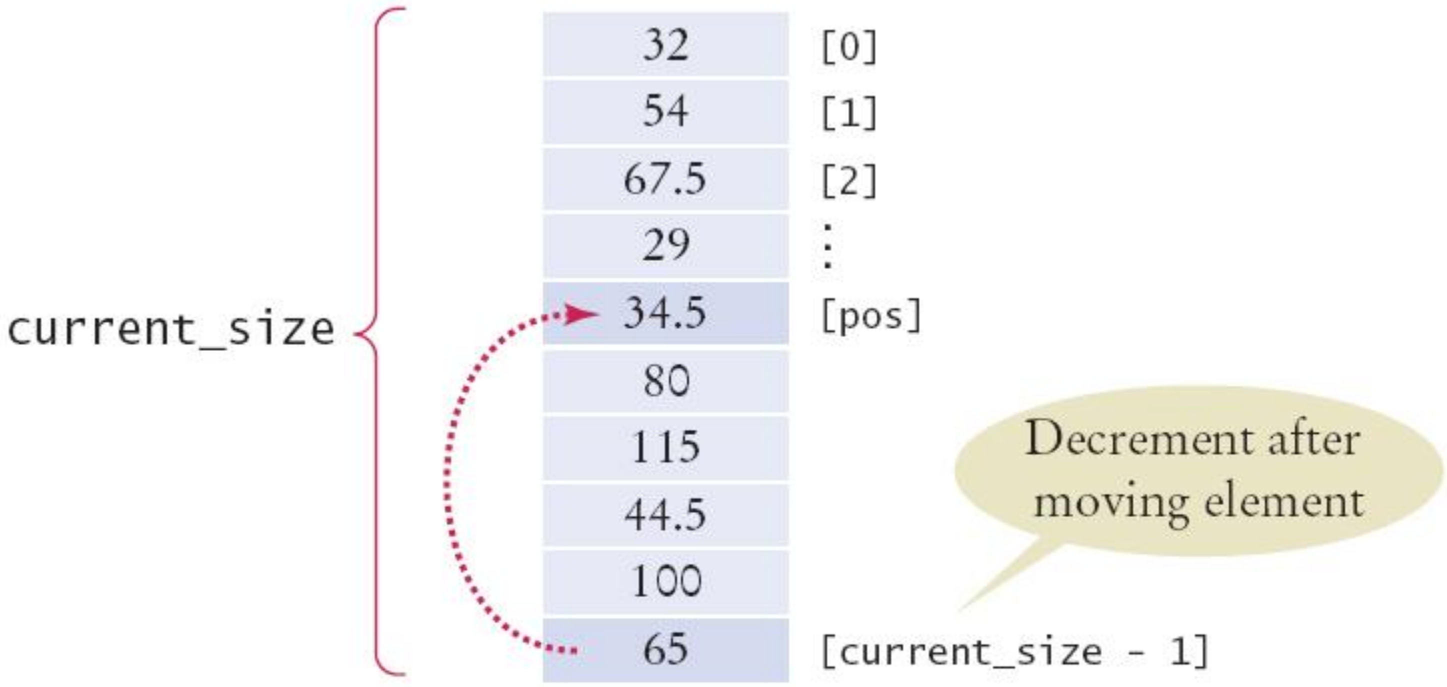
If the elements in the array are not in any particular order, that task is easy to accomplish.

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

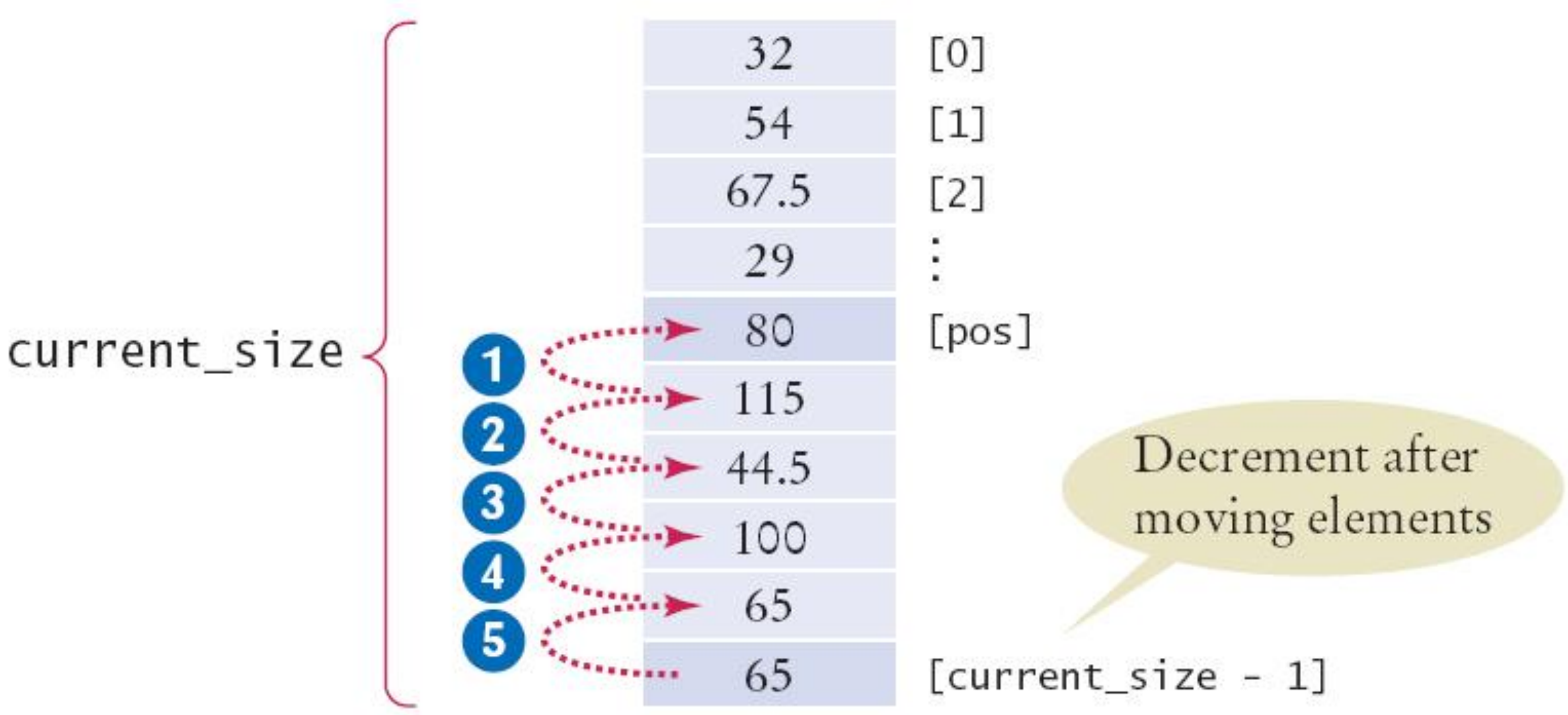
```
values[pos] = values[current_size - 1];

current_size--;
```



Common Algorithms – Removing an Element, Ordered

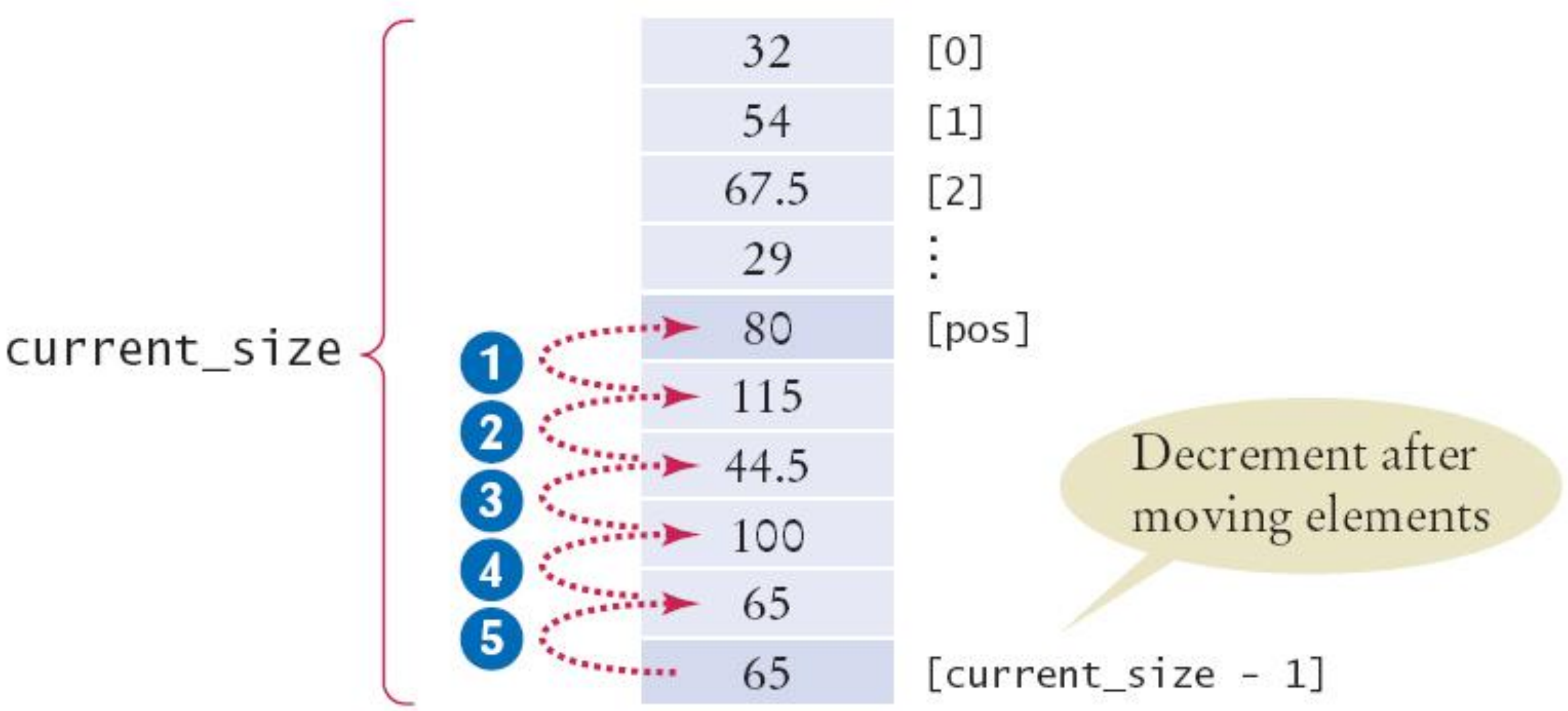
The situation is more complex
if the order of the elements matters.

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.



Common Algorithms – Removing an Element, Ordered

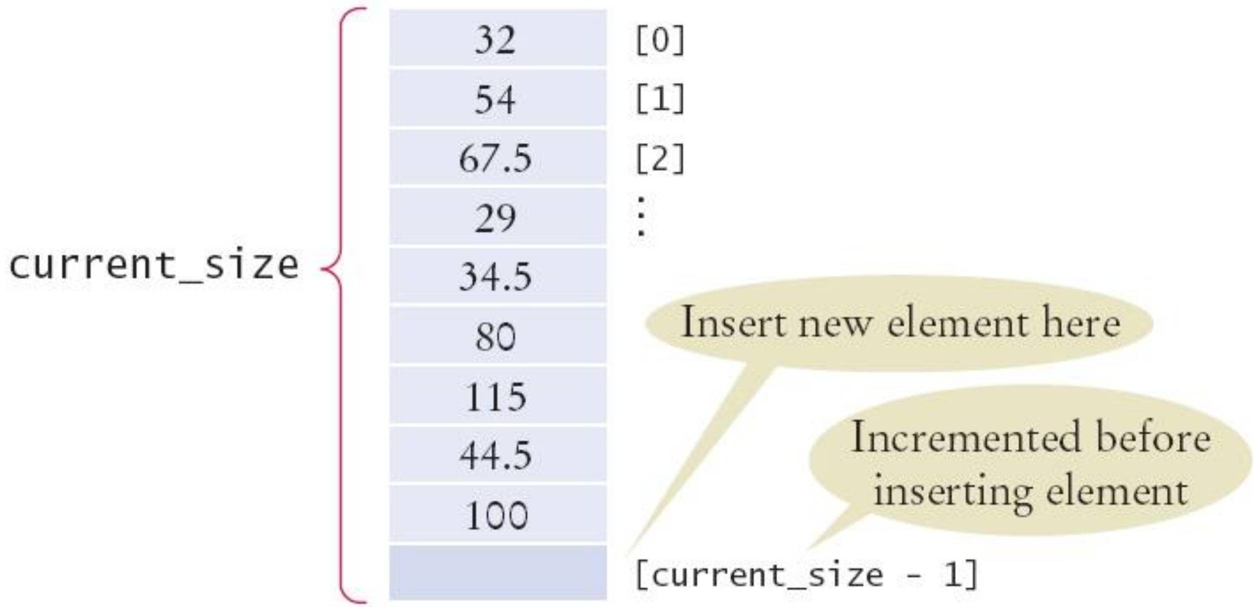
```
for (int i = pos + 1; i < current_size; i++) {
   values[i - 1] = values[i];
}
current_size--;
```



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;
}
```

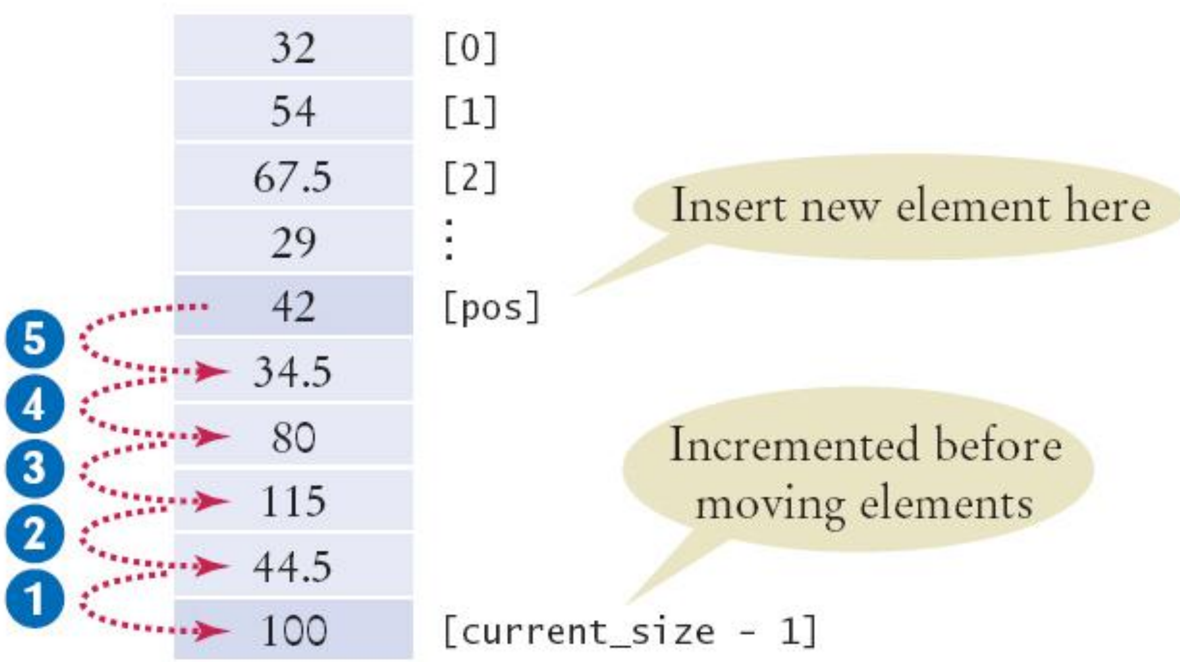


Common Algorithms – Inserting an Element Ordered

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

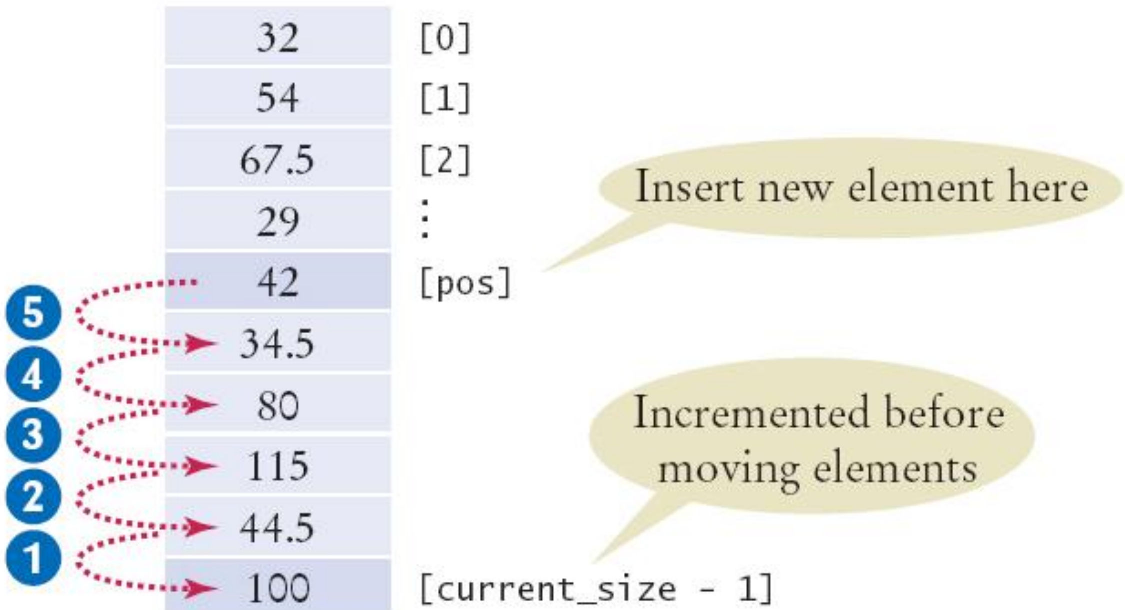
First, you must make the array one larger by incrementing `current_size`.

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++) {
   scanf("%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 && 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.

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 && 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;
}
```

Example: Printing a Histogram

```c
/* fig06_08.c
   Histogram printing program */
#include <stdio.h>
#define SIZE 10

int main() {
   // use initializer list to initialize array n
   int n[SIZE] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
   int i; // outer for counter for array elements
   int j; // inner for counter counts stars in each histogram bar

   printf( "Element  Value  Histogram \n" );

   // for each element of array n, output a bar of the histogram
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%7d %13d        ", i, n[i]) ;

      for ( j = 1; j <= n[i]; j++ ) {   // print one bar
         printf( "*" );
      } // end inner for

      printf( "\n" ); // end a histogram bar
   } // end outer for
} // end main
```

Program Output

```
Element        Value        Histogram
      0           19        *******************
      1            3        ***
      2           15        ***************
      3            7        *******
      4           11        ***********
      5            9        *********
      6           13        *************
      7            5        *****
      8           17        *****************
      9            1        *
```

Histograms are printed horizontally.

Example: Statistical Calculations

- Read student scores entered by user, and store into an array X.
- Calculate the followings.

$$Average\ (\bar{x}) = \frac{\sum_{i=1}^{N} X_i}{N} \qquad Variance = \frac{\sum_{i=1}^{N} (x_i - \bar{x})^2}{N}$$

$$\text{Standard deviation} = \sqrt{Variance}$$

$$\text{Absolute deviation} = \frac{\sum_{i=1}^{N} \left| x_i - \bar{x} \right|}{N}$$

Example: Statistical Calculations

Part 1 of 2

```
#include <stdio.h>
#include <math.h>  // pow,fabs,sqrt functions
#define MAXSTUDENTS 100

int main()
{
    int score[MAXSTUDENTS]; // Array
    int N = 0; // Number of students
    float avg, variance, std_dev, abs_dev;
    float total = 0, sqr_total = 0, abs_total = 0;
    int i = 0;

    printf("How many students are there ? ");
    scanf("%d", &N);

    for (i = 0; i < N; i++)
    {
        printf("Enter grade of student # %d : ", i + 1);
        scanf("%d", &score[i]);
        total += score[i];
    }
```

Example: Statistical Calculations

Part 2 of 2

```
    avg = total / N;

    for (i = 0; i < N; i++)
    {
        sqr_total += pow( score[i] - avg , 2);
        abs_total += fabs(score[i] - avg);
    }

    variance = sqr_total / N;
    std_dev = sqrt(variance);
    abs_dev = abs_total / N;

    printf("Average            =  %f\n", avg);
    printf("Variance           =  %f\n", variance);
    printf("Standard deviation =  %f\n", std_dev);
    printf("Absolute deviation =  %f\n", abs_dev);

} // end main
```

Program
Output

```
How many students are there ? 4

Enter grade of student # 1 :   92
Enter grade of student # 2 :   62
Enter grade of student # 3 :   70
Enter grade of student # 4 :   51


Average             =  68.750000
Variance            =  300.916656
Standard deviation =  17.346949
Absolute deviation =  12.250000
```