© traveler1116/iStockphoto.

Arrays

Slides by Evan Gallagher

Using Arrays

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

Using Arrays

In arrays, the stored data is of

the *same* type

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)

Example Task with Several Numbers

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.)

Problem: Each Number as a Separate Variable Name

32 54 67.5 29 35 80 115 44.5 100 65

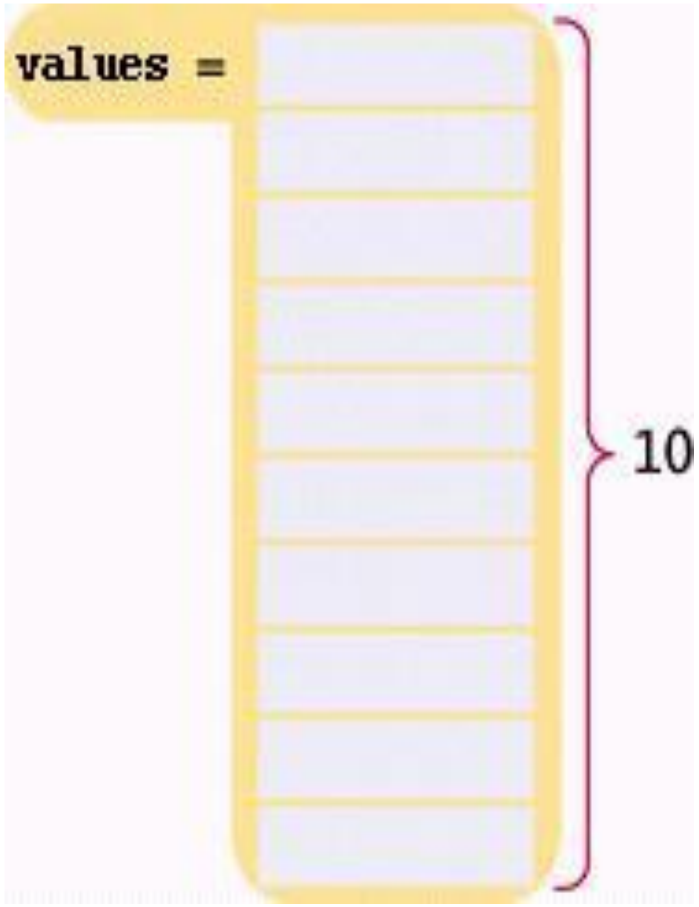
So you would create a variable for each,
of course!

```
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
Arrays store data with a single name and a subscript, like in math vectors.



We can declare an array as:

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

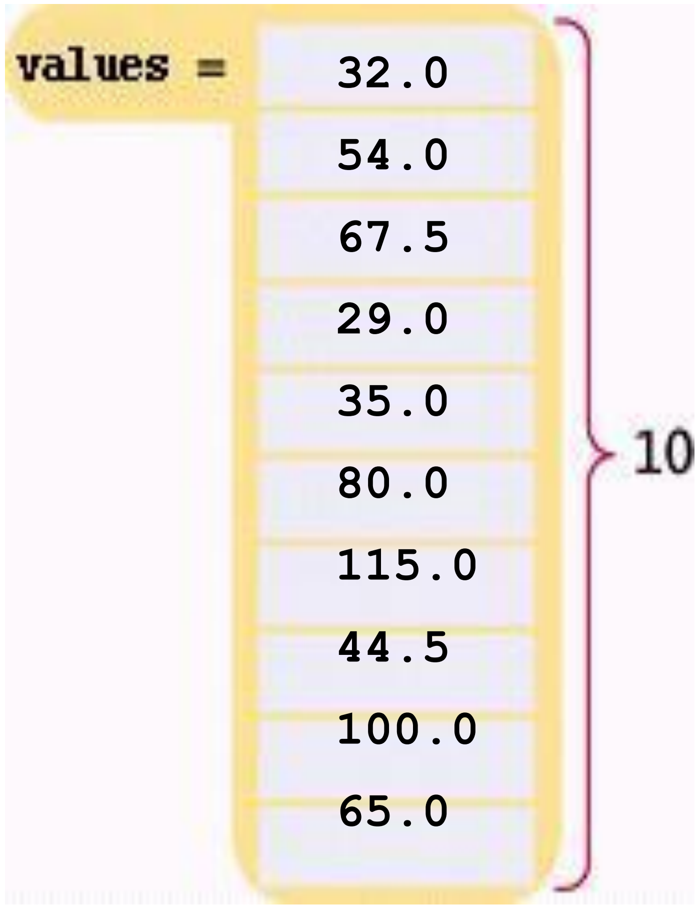
An "array of double"

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

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 Examples: Table 1

`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: the size must be a constant. This code 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*.

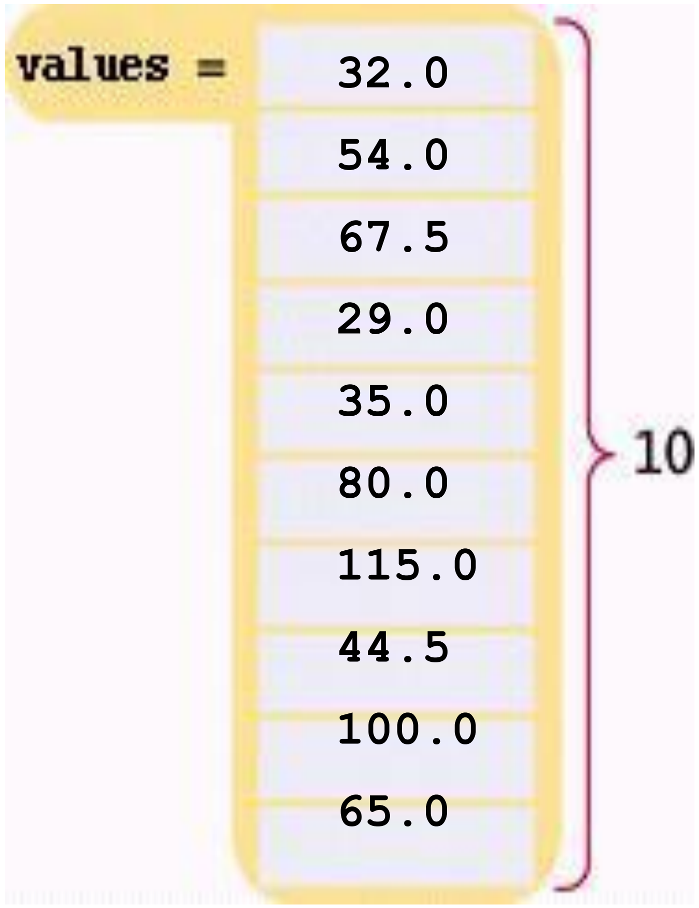
The first element in the array is at index `i`=0, *NOT at `i`=1.*

Array Element Index

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`.

(Again because the first subscript is 0, the output for index=4 is the 5th element)

Array Element Index for Writing

The same notation can be used to change the element.

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

Array Element Indices are between 0 and Length-1

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 – Capacity

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

We call this quantity the *capacity*.

For example, we may decide a problem usually needs 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 – Current Size

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

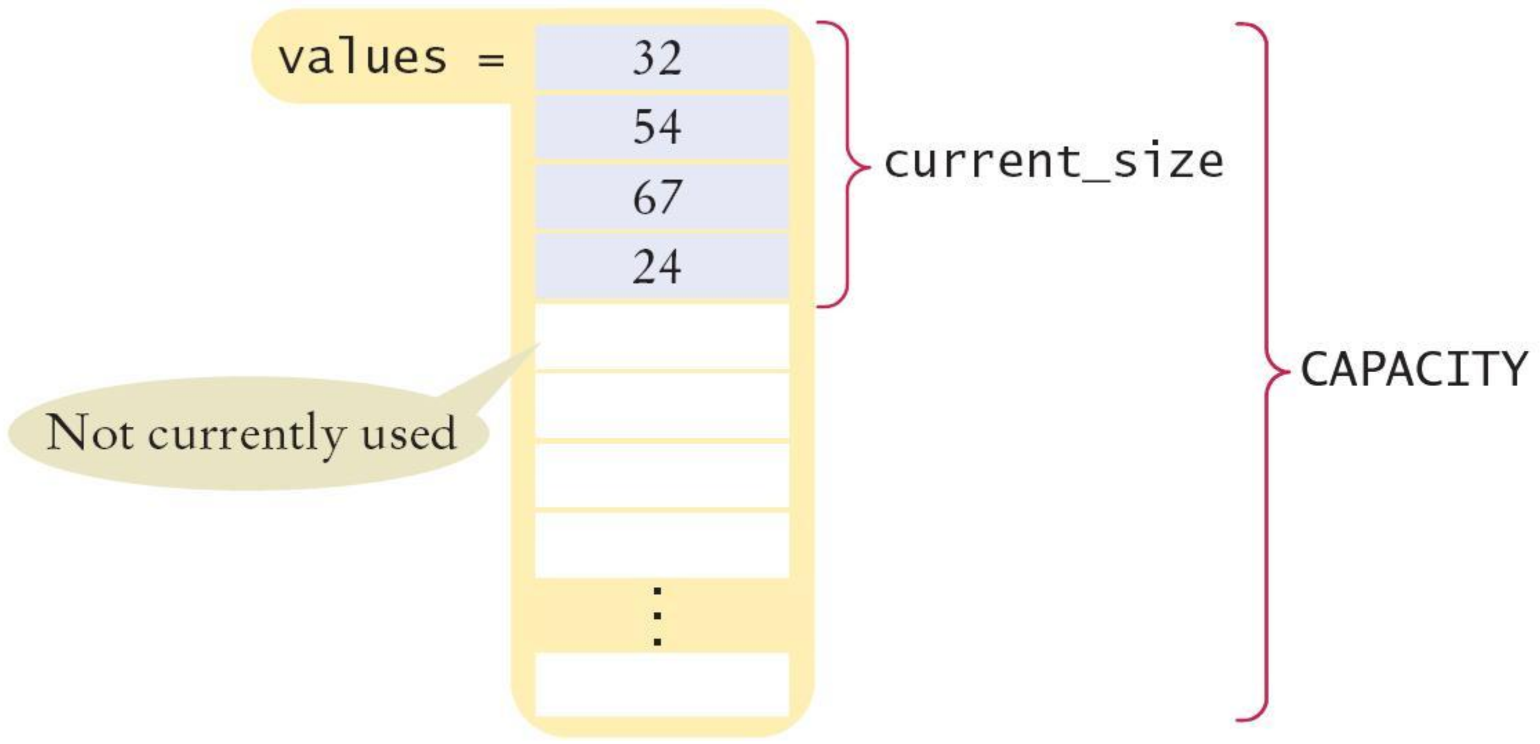
We will use a *companion variable* to hold that amount:

```
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-Filling an Array – Code Loop

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

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

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

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

Partially-Filled Arrays – Output

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;
}
```

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 `for` loop, whose counter is the array index:

```
 const int CAPACITY =10;
 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 tenth and ***last legal*** element.

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]; //error! 9 is the last valid index
```

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

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];
```

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!