Arrays

Arrays

- What are arrays?
- Initializing arrays
- Common array algorithms

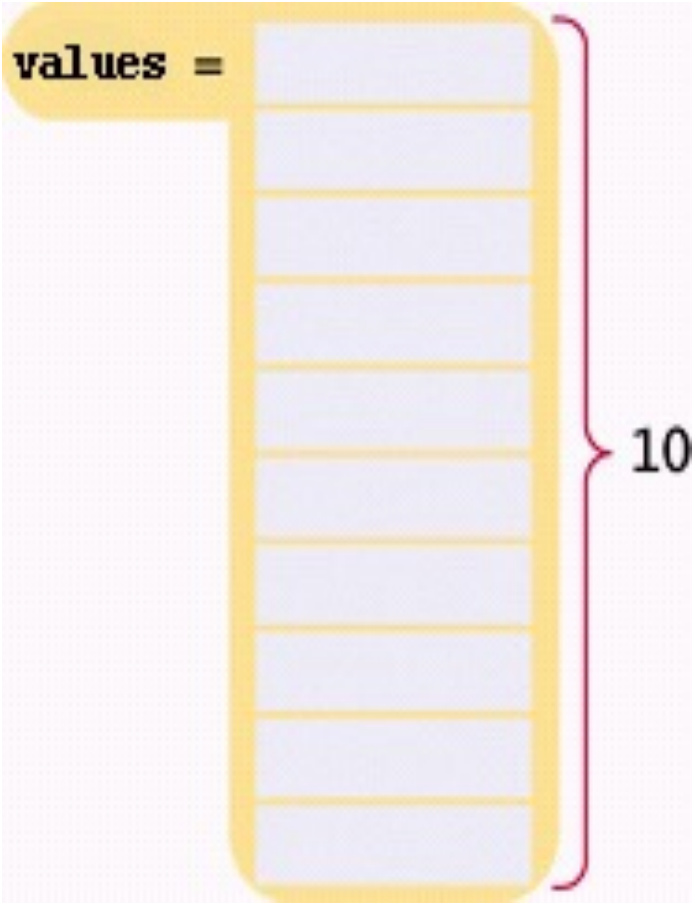
Using Arrays

32 54 67.5 29 35 80 115 44.5 100 65

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

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

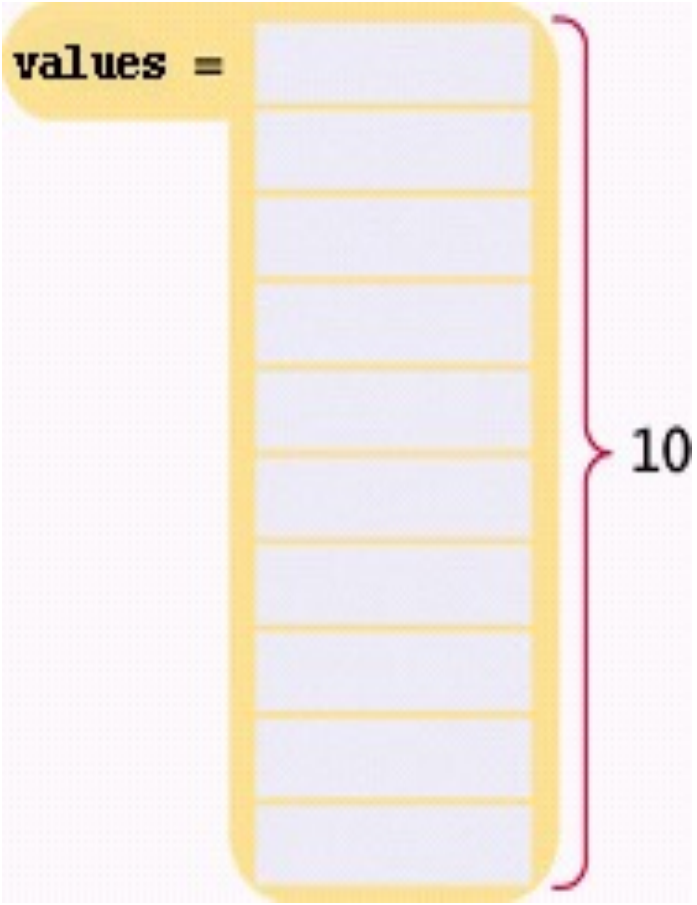
Using Arrays



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

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

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

Array Syntax

Defining an Array

Size must be a constant.

Ok to omit size if initial values are given.

Element type Name Size

```
double values[5] = { 32, 54, 67.5, 29, 34.5 };
```

Optional list of initial values

Use brackets to access an element.

```
values[i] = 0;
```

The index must be ≥ 0 and < the size of the array.

Introduction to Arrays

Definition: An array is a collection of data of the same type, referenced as different elements of the same name.

- First "aggregate" data type
  - Means "grouping"
  - *int, float, double, char* are simple data types
- Used for lists of like items
  - Test scores, temperatures, names, etc.
  - Avoids declaring multiple simple variables
  - Can manipulate "list" as one entity

Declaring Arrays

Declare the array → allocates memory

```
int score[5];
```

- Declares array of 5 integers named "score"
- Similar to declaring five variables:

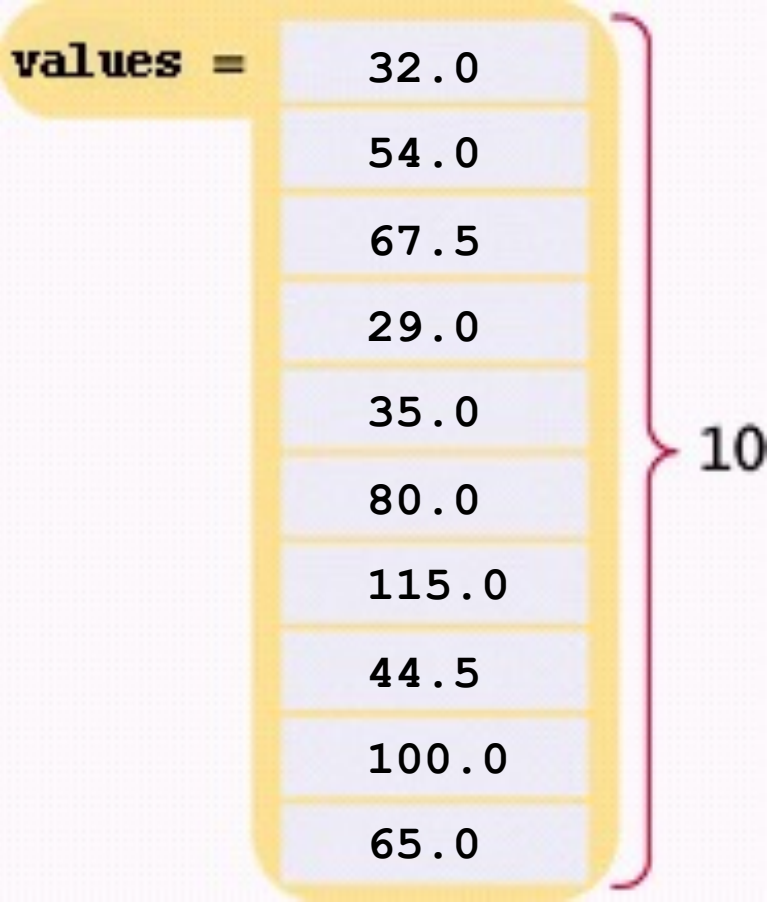
```
int score[0], score[1], score[2], score[3], score[4];
```

- Individual parts can be called many things:
  - Indexed or subscripted variables
  - "Elements" of the array
  - Value in brackets is called index or subscript
  - Numbered from 0 to (size – 1)

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 Arrays

- Access using index/subscript

```
cout << score[3];
```

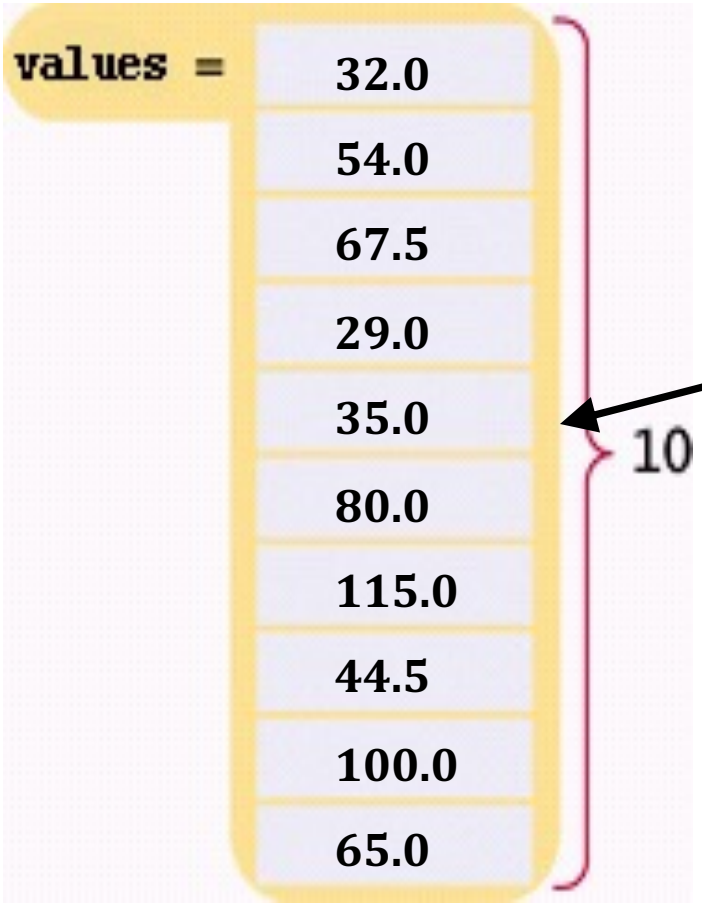
- Note two uses of brackets:
  - In declaration, specifies SIZE of array
  - Anywhere else, specifies a subscript
- Size, subscript need not be literal

```
int score[MAX_SCORES];
score[n+1] = 99;  -->
```

If n is 2, identical to: score[3]

Accessing an Array Element

The same notation can be used to change the element.



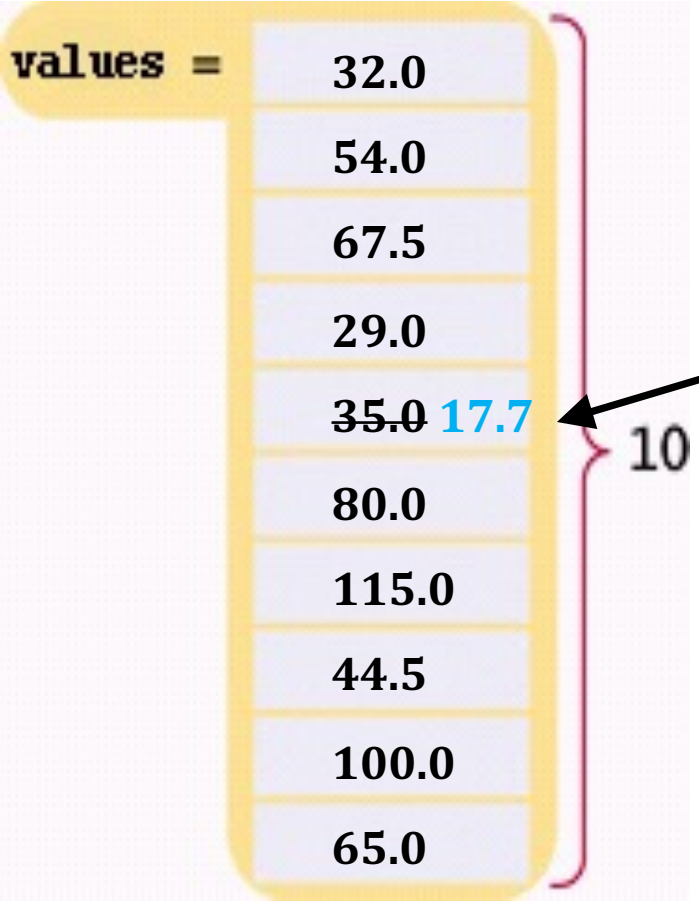

```
double values[10];
...
cout << values[4] << endl;
```

The output will be **35.0**.

Accessing an Array Element

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

4 is the *index*.




```
values[4] = 17.7;

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

The output will be **17.7**.

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 or < 10**

0, 1, 2, 3, 4, 5, 6, 7, 8, 9 is … 10 numbers.

Common Array Algorithms

Array Usage

- Powerful storage mechanism
- Can issue commands like:
  - "Do this to i^{th} indexed variable”, where i is computed by program
  - "Display all elements of array score"
  - "Fill elements of array score from user input"
  - "Find highest value in array score"
  - "Find lowest value in array score”
- Disadvantages: size MUST BE KNOWN at declaration

Common Algorithms – Filling

- This loop fills an array with zeros:

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

- To fill an array with squares (0, 1, 4, 9, 16, ...).

```
for (int i = 0; i < size; 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?
- Let's try what seems right and easy…
  - `squares = lucky_numbers;`
    …and wrong!
  - *You cannot assign arrays!*
  - *The compiler will report a syntax error.*

Common Algorithms – Copying Requires a Loop

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

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

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

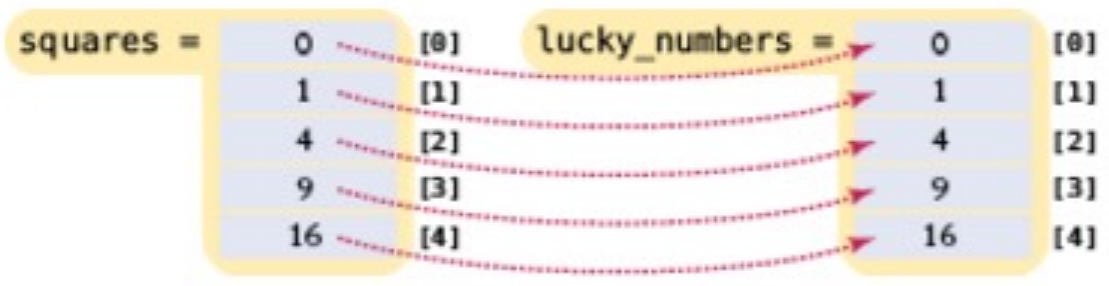


Figure 4 Copying Elements to Copy an Array

Common Algorithms – Sum and Average Value

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

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

The average is just arithmetic:

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

Common Algorithms – Maximum

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; i++)
{
    if (values[i] > largest)
    {
        largest = values[i];
    }
}
```

Common Algorithms – Minimum

For the minimum, we just reverse the comparison.

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

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

Common Algorithms – Linear Search

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

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

Common Algorithms – Swapping Elements

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

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

- Look closely! In the first line you lost – forever! – the value at i, replacing it with the value at j.
- Then what?
- Put' j's value back in j in the second line?
- We end up with 2 copies of the [j] value, and have lost the [i]

Code for Swapping Array Elements

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

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

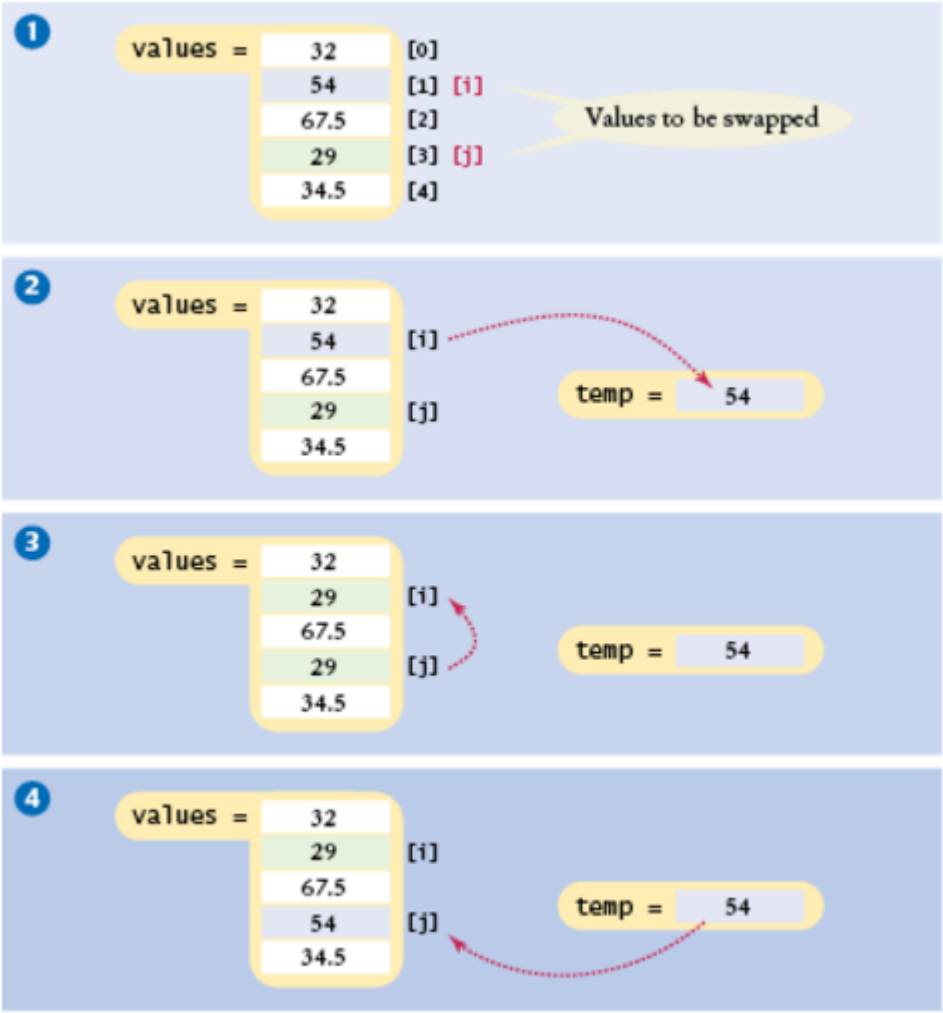


Figure 9 Swapping Array Elements

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 Unknown # of Inputs

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 current_size variable will have the number of inputs.

```
double values[CAPACITY];
int current_size = 0;
double input;
while (cin >> input) //cin returns true until
  // invalid (non-numeric) char encountered
{
       if (current_size < CAPACITY)
       {
          values[current_size] = input;
          current_size++;
       }
}
```

Complete Program to Read Inputs and Report the Maximum

```
#include <iostream>
using namespace std;

int main() //read inputs, print out largest
{
   const int CAPACITY = 1000;
   double values[CAPACITY];
   int current_size = 0;

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

Complete Program to Read Inputs and Report the Maximum Part 2

```
double largest = values[0];
   for (int i = 1; i < current_size; i++)
   {
      if (values[i] > largest)
      {
         largest = values[i];
      }
   }
   for (int i = 0; i < current_size; i++)
   {  //print each element, highlighting largest
      cout << values[i];
      if (values[i] == largest)
      {
         cout << " <== largest value";
      }
      cout << endl;
   }
   return 0;
}
```

pass by reference

cup =

fillCup( )

pass by value

cup =

fillCup( )

www.mathwarehouse.com