© traveler1116/iStockphoto.

Chapter Six: Arrays and Vectors

Slides by Evan Gallagher

Chapter Goals

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

Topic 2

1. Arrays
2. Common array algorithms
3. Arrays / functions
4. Problem solving: adapting algorithms
5. Problem solving: discovering algorithms
6. 2D arrays
7. Vectors
8. Chapter Summary

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; 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;
}
```

Self-Check Exercise

- Using a `for` loop, fill an array a with 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2
  - Hint: you'll need to increment the loop index by 3, not by 1

```
for (          ;          ;          )
   {
      a[       ] =   ;
      a[       ] =   ;
      a[       ] =   ;
   }
```

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];
}
```

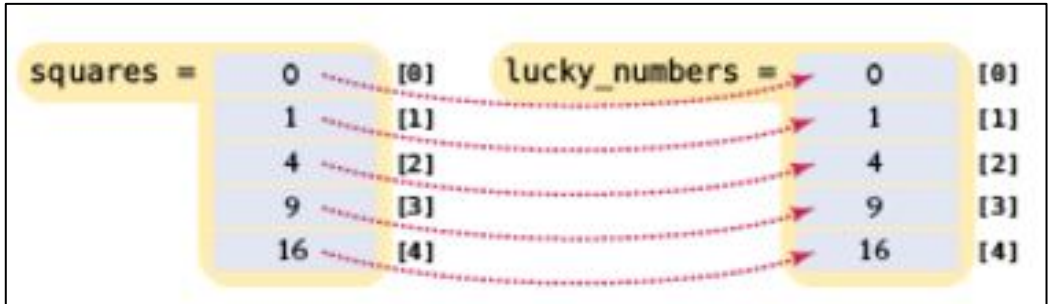

Figure 4 Copying Elements to Copy an Array

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 – Element Separators

When you display the elements of a vector, you may 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 Separator Code

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

Common Algorithms – Removing an Element, Unordered

To remove the element at index `i`:

If the elements in the array are not in any particular order, simply overwrite the element to be removed with the *last* element, then shrink the size by 1.

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

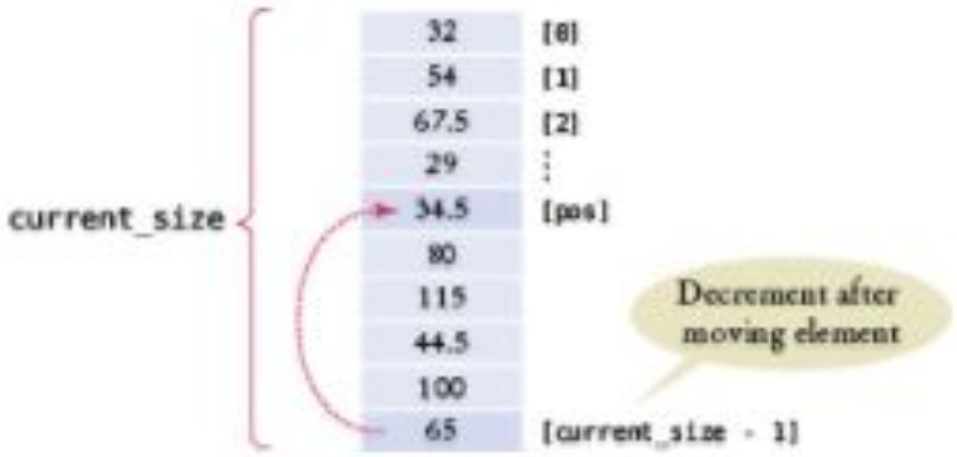

Figure 5 Removing an Element in an Unordered Array

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

```
//removing the element at index "pos"
for (int i = pos + 1; i < current_size; i++)
{
   values[i - 1] = values[i];
}
current_size--;
```

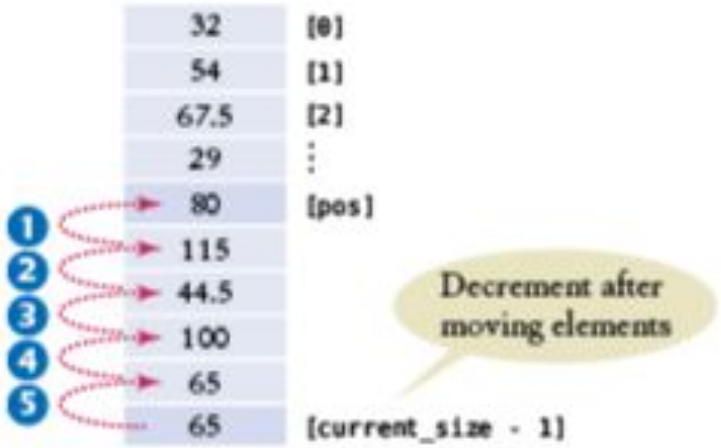

Figure 6 Removing an Element in an Ordered Array

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 "out" to higher indices.

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

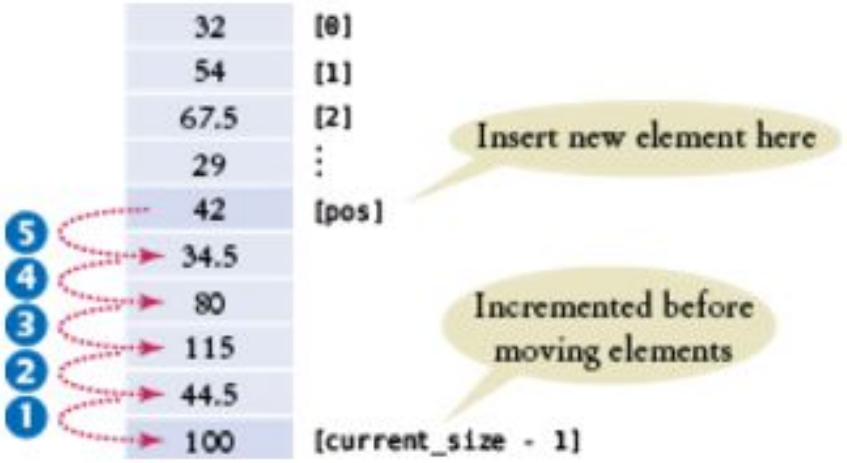

Figure 8 Inserting an Element in an Ordered Array

Inserting an Element Ordered: Code

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

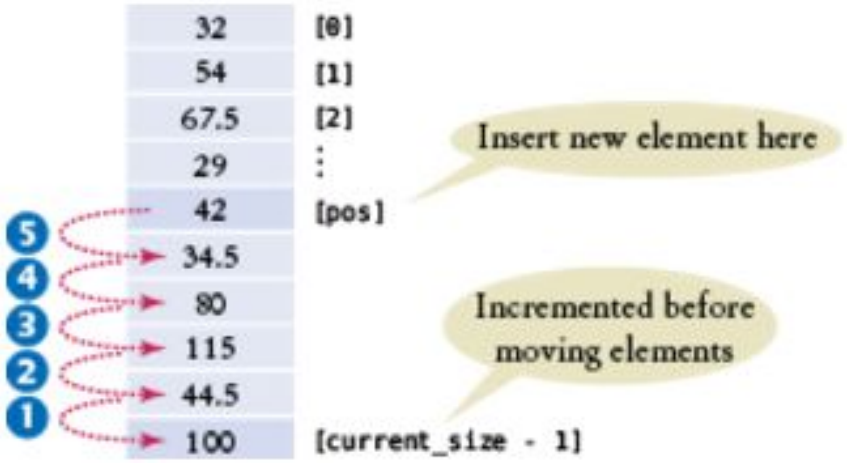

Figure 8 Inserting an Element in an Ordered Array

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 `[j]`, 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;
```

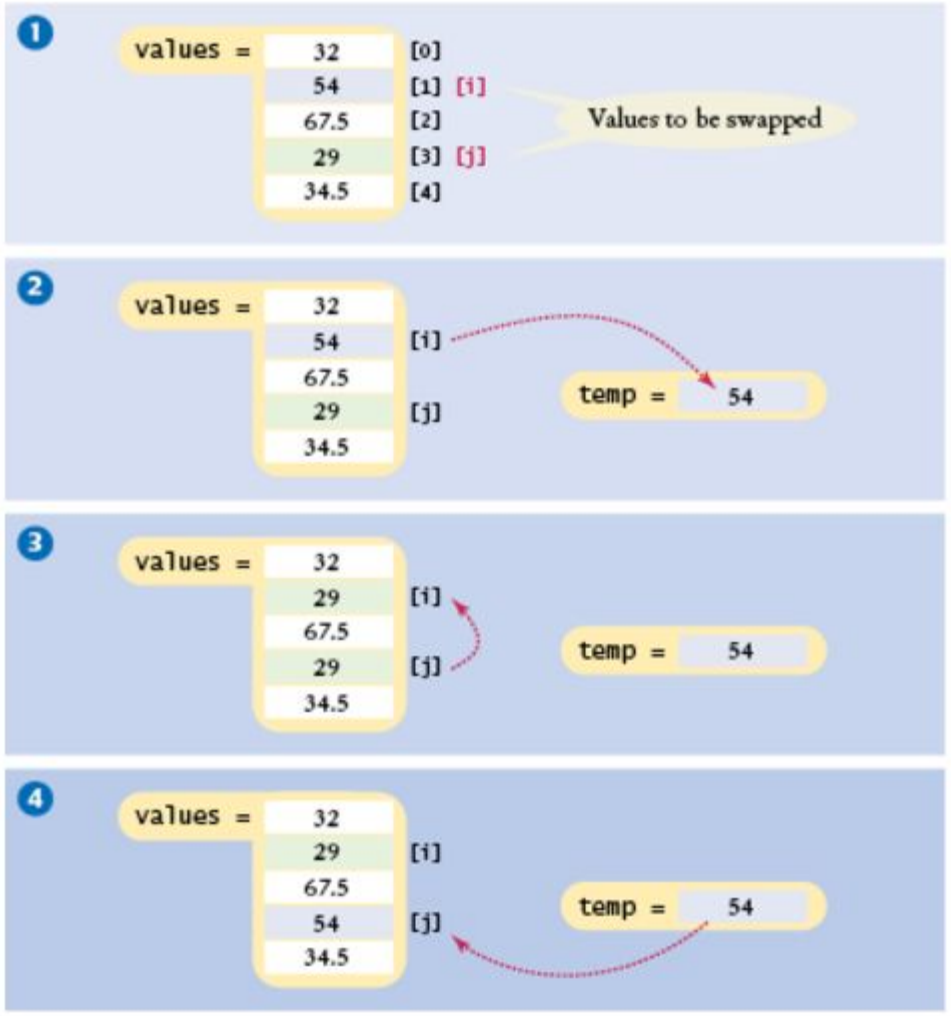

Figure 9 Swapping Array Elements

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, **`current_size`** will have the number ofinputs.

```
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++;
   }
}
```

Common Algorithms – Overflow 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.

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, 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;
}
```

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: the `sort` function

To sort the elements into ascending numerical order, you can call the **`sort`** library function from the `<algorithm>` library.

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

```
#include <algorithm>
...
int main()
{
   ...
   sort(values, values + current_size);
```

A Sorting Algorithm: Selection Sort

The following is an inefficient but simple sorting algorithm. It divides the array into a sorted section on the left and unsorted on the right, moving elements successively from right to left starting with the smallest element remaining in the unsorted section. The `sort()` function in the library is a much faster algorithm, but here is the *selection sort*:

```
for (int unsorted = 0; unsorted < size - 1; unsorted++)
{
   // Find the position of the minimum
   int min_pos = unsorted;
   for (int i = unsorted + 1; i < size; i++)
   {
      if (values[i] < values[min_pos]) { min_pos = i; }
   }
   // Swap the minimum into the sorted area
   if (min_pos != unsorted)
   {
      double temp = values[min_pos];
      values[min_pos] = values[unsorted];
      values[unsorted] = temp;
   }
}
```

In a later chapter we'll cover sorting algorithms in detail.

Searching Algorithms: Binary Search

- There is a much faster way to search a sorted array than the linear search shown previously
- "Binary search" repeatedly partitions the array in half, then ¼, then 1/8, etc…to find a match

```
bool found = false;
int low = 0, high = size - 1;
int pos = 0;
while (low <= high && !found)
{
   pos = (low + high) / 2; // Midpoint of the subarray
   if (values[pos] == searched_value)
   { found = true; }
   else if (values[pos] < searched_value)
      { low = pos + 1; } // Look in second half
   else { high = pos - 1; } // Look in first half
}
if (found)
   { cout << "Found at position " << pos; }
```