© traveler1116/iStockphoto.

Chapter Six: Arrays and Vectors

Slides by Evan Gallagher

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Chapter Goals

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Topic 1

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Using Arrays

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Using Arrays and Vectors

In both vectors and 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)

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.)

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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!

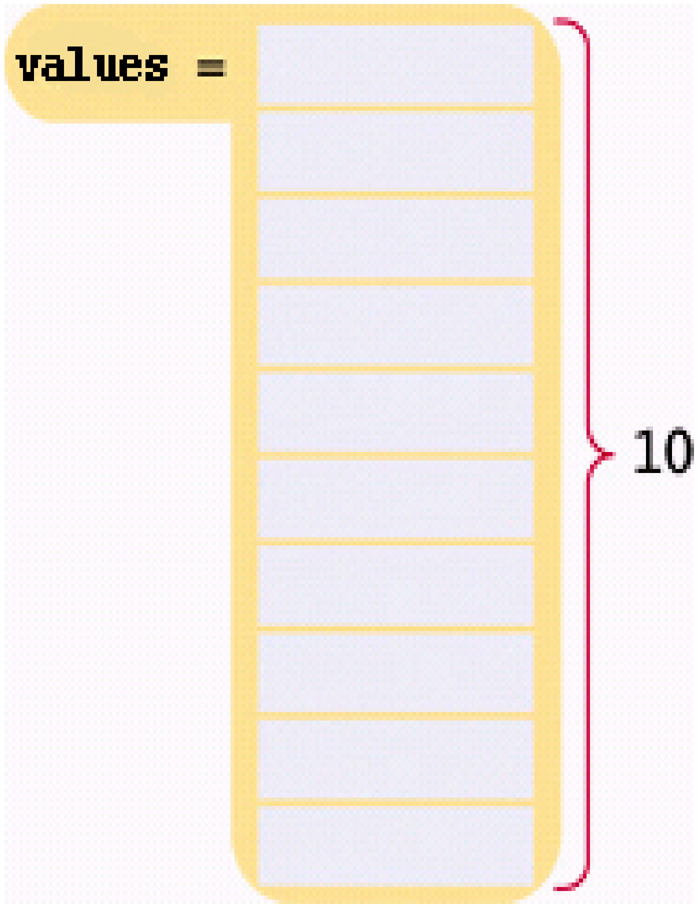
```
int n1, n2, n3, n4, n5, n6, n7, n8, n9, n10;
```

Then what ???

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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"

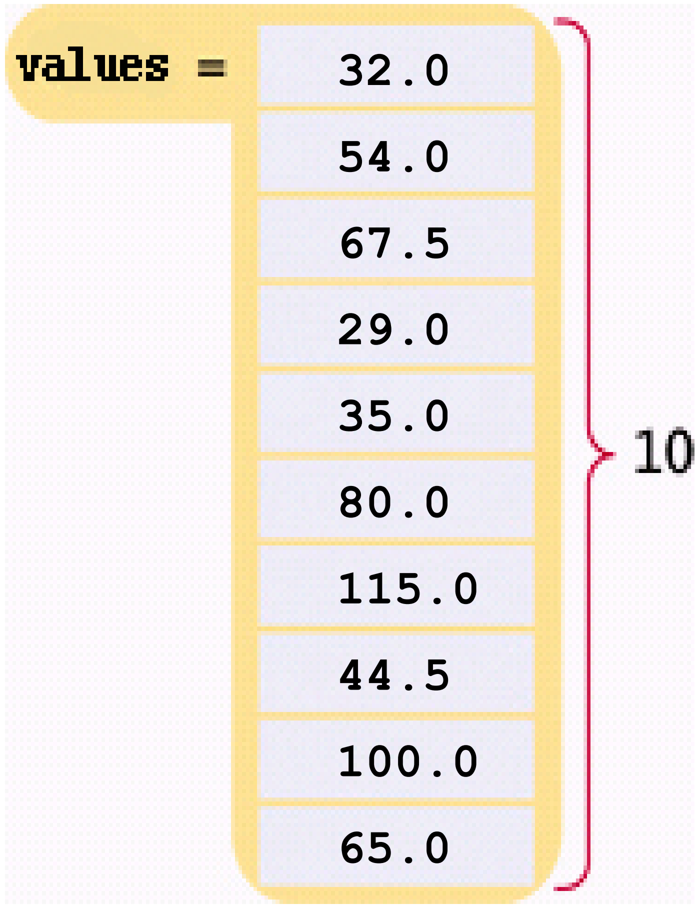
Ten elements of double type stored under one name as an array.

Copyright © 2018 by John Wiley & Sons. All rights reserved

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 };
```



Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Copyright © 2018 by John Wiley & Sons. All rights reserved

Accessing an Array Element

An array element can be used like any variable.

To access an array element, you use the notation:

`values[i]`

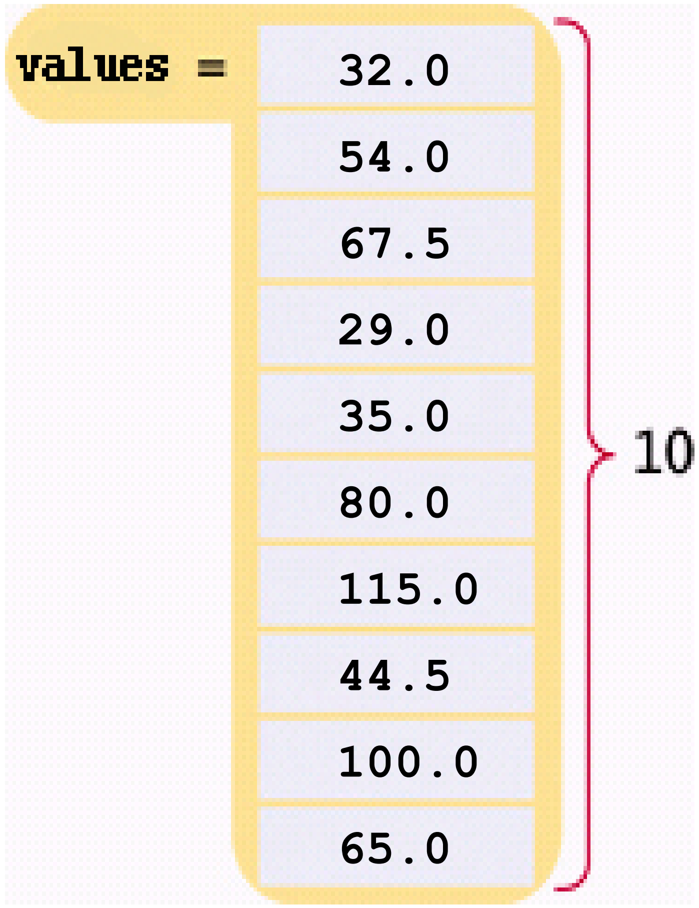
where **`i`** is the *index*.

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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)

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Array Element Index for Writing

The same notation can be used to change the element.

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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];
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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:

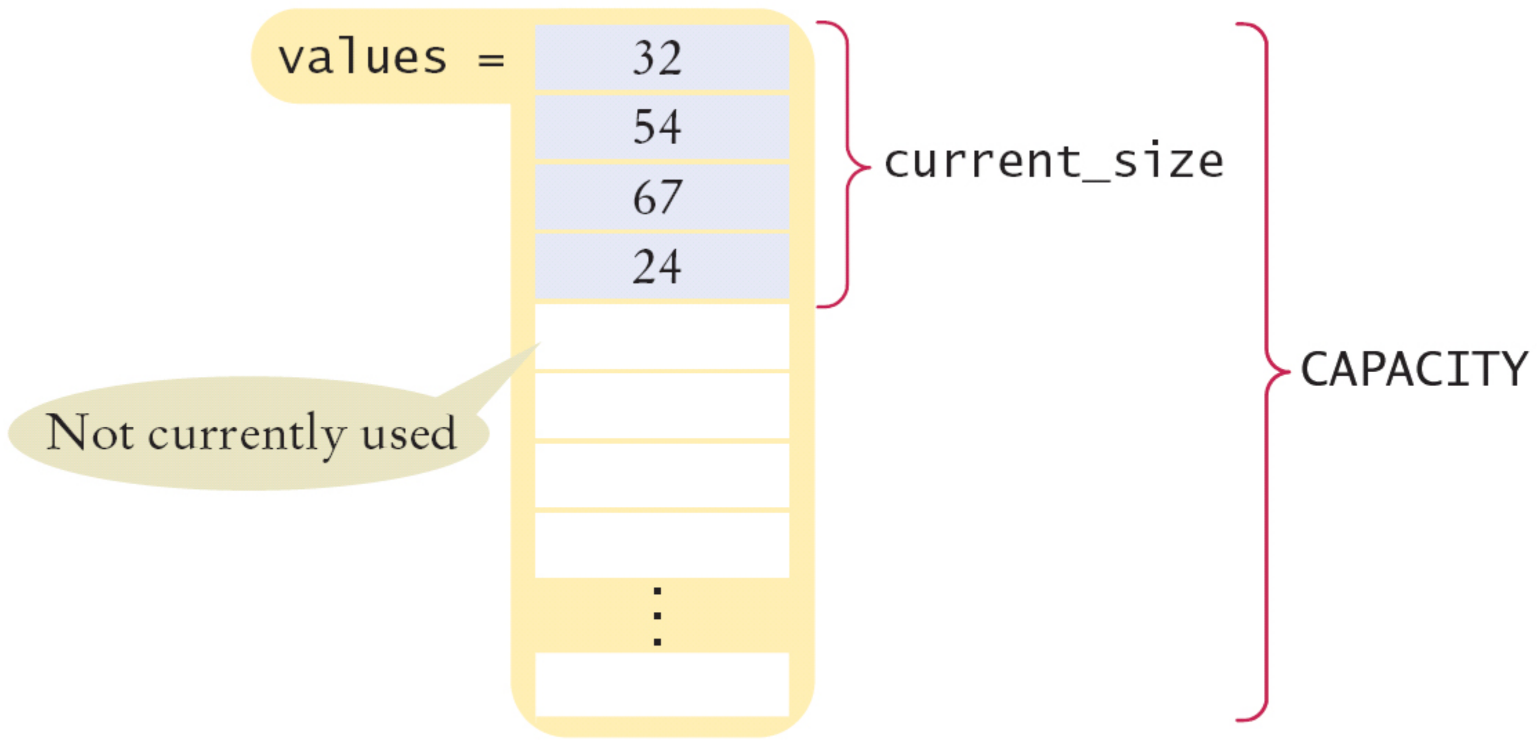
```
const int CAPACITY = 100;
double values[CAPACITY];

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

Suppose we add four elements to the array?

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Partially-Filled Arrays – Companion Variable for Size

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

current_size = 4; // array now holds 4
```



Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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!

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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)

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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[        ] =   ;
   }
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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];
}
```

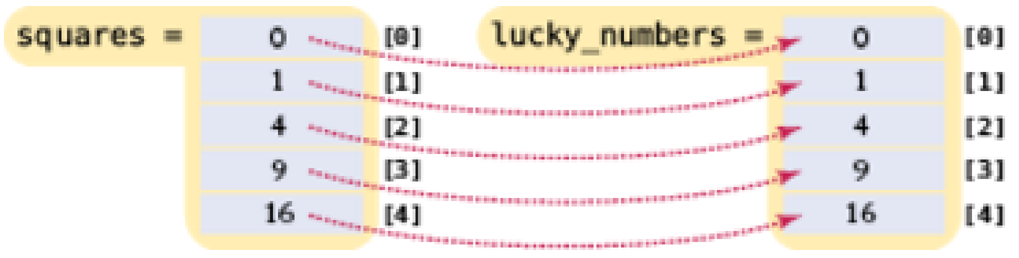


Figure 4 Copying Elements to Copy an Array

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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;
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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];
   }
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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];
}
```

Copyright © 2018 by John Wiley & Sons. All rights reserved

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++;
   }
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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--;
```

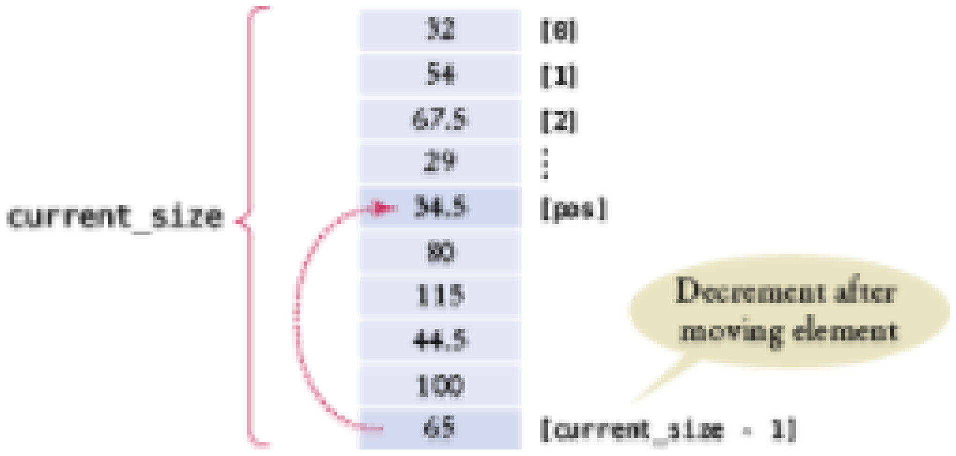


Figure 5 Removing an Element in an Unordered Array

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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--;
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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--;
```

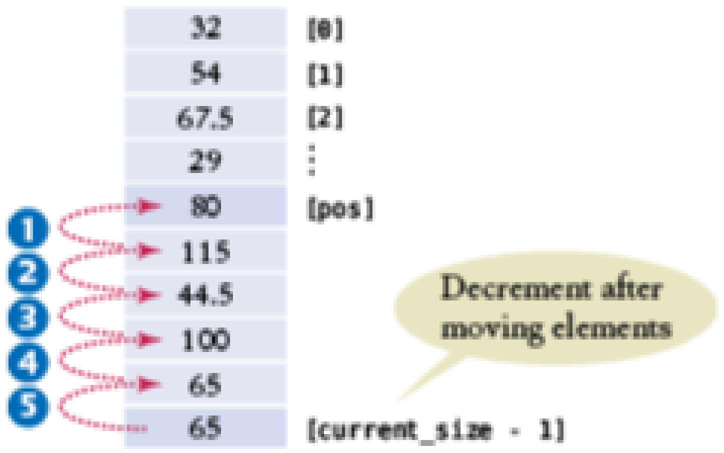


Figure 6 Removing an Element in an Ordered Array

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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]`**.

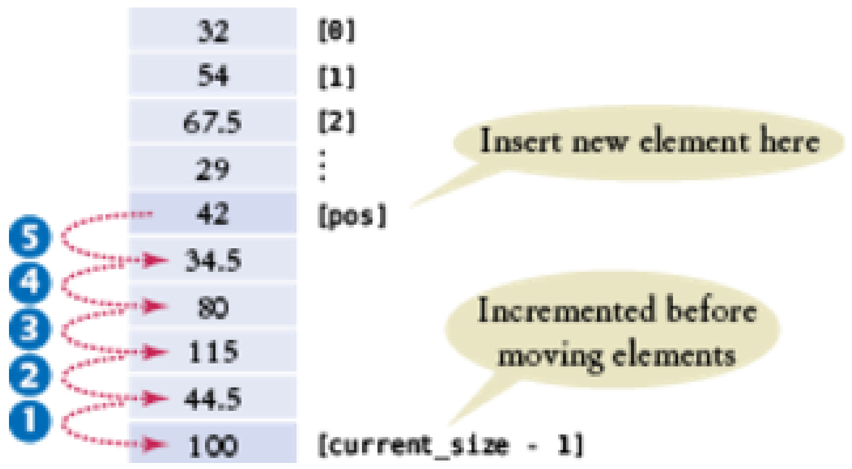


Figure 8 Inserting an Element in an Ordered Array

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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;
}
```

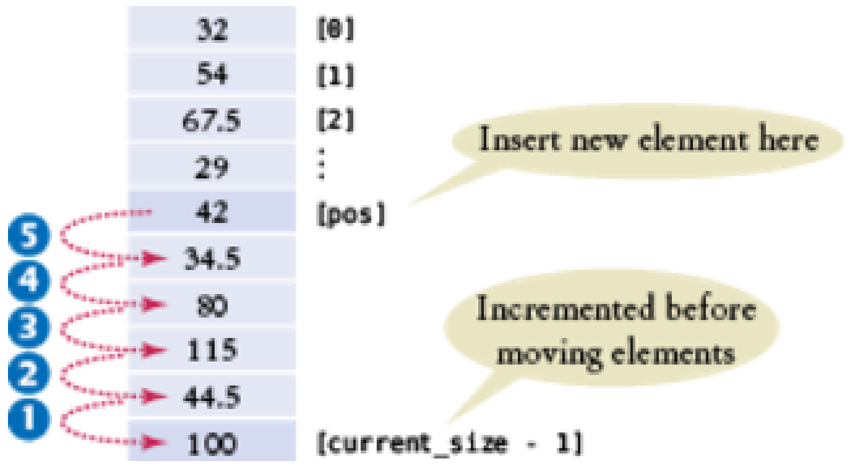


Figure 8 Inserting an Element in an Ordered Array

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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]`

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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;
```

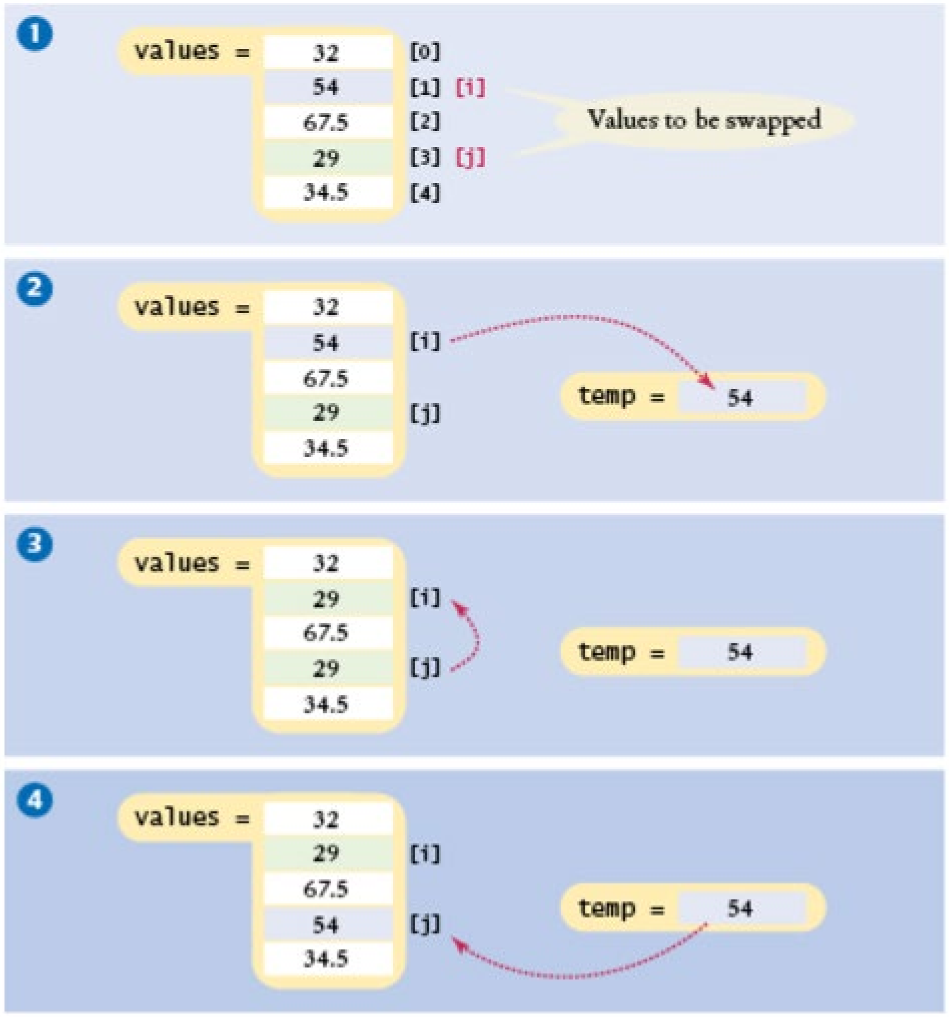


Figure 9 Swapping Array Elements

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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];
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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++;
    }
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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++;
      }
   }
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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);
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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; }
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Topic 3

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Array Parameter Function Example

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;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Array Parameters in Functions Require `[]` in the Header

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)
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Array Function Call Does NOT Use the Brackets!

When you call the function, supply both the name of the array and the size, BUT NO SQUARE BRACKETS!!

```
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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Array Parameters Always are Reference Parameters

When you pass an array into a function, the contents of the array can ***always*** be changed. An array name is actually a reference, that is, a memory address:

```
   //function to scale all elements in array by a factor
void multiply(double values[], int size, double factor)
   {
      for (int i = 0; i < size; i++)
      {
         values[i] = values[i] * factor;
      }
   }
```

But never use an & with an array parameter – that is an error.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Arrays as Parameters but No Array Returns

You can pass an array into a function
but

you cannot return an array.

However, the function can modify an input array, so the function definition must include the result array in the parentheses if one is desired.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Arrays as Parameters and Return Value

If a function can change the size of an array,
it should let the caller know the new size by returning it:

```
int read_inputs(double inputs[], int capacity)
{ //returns the # of elements read, as int
   int current_size = 0;
   double input;
   while (cin >> input)
   {
      if (current_size < capacity)
      {
         inputs[current_size] = input;
         current_size++;
      }
   }
   return current_size;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Array Parameters in Functions: Calling the Function

Here is a call to the **`read_inputs`** function:

```
const int MAXIMUM_NUMBER = 1000;
double values[MAXIMUM_NUMBER];
int current_size =
  read_inputs(values, MAXIMUM_NUMBER);
```

After the call, the **`current_size`** variable specifies how many were added.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Function to Fill or Append to an Array

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++;
      }
   }
}
```

Note this function has the added benefit of either filling an empty array or appending to a partially-filled array.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Array Functions Example

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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Array Functions Example Code, Part 1

```
#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)
   {
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Array Functions Example Code, part 2

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Array Functions Example Code, part 3

```
/**
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;
   }
}
/**
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;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Array Functions Example Code, part 4

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

   return 0;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Constant Array Parameters

- When a function doesn’t modify an array parameter, it is considered good style to add the `const` reserved word, like this:

```
double sum(const double values[], int size)
```

- The `const` reserved word helps the reader of the code, making it clear that the function keeps the array elements unchanged.

- If the implementation of the function tries to modify the array, the compiler issues a warning.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Topic 4

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Problem Solving: Adapting Algorithms

Recall that you saw quite a few
(too many?)
algorithms for working with arrays.

Suppose you need to solve a problem that
does not exactly fit any of those?

What to do?

No, "give up" is not an option!

You can adapt algorithms you already know to produce a new algorithm.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Problem Example: Summing Quiz Scores

Consider this problem:

Compute the final quiz score from a set of quiz scores,

but be nice:

drop the lowest score.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Adapting Algorithms: Three that We Know

Calculate the sum:

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

Find the minimum:

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

Remove an element:

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Adapting Algorithms: A Glitch in Combining Those Three

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

This algorithm removes by knowing *the position* of the element to remove…

…but…

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

That's not the *position* of the smallest – it IS the smallest.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Algorithm to Find the Position

Here's another algorithm I know that *does* find the position:

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Adapting the Minimum Algorithm to Report the Position

Combining the minimum value algorithm with the position-finder:

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Final Answer for Adapting Algorithms

Aha! Here is the algorithm:

1. *Find the **position** of the minimum*
2. *Remove it from the array*
3. *Calculate the sum*
   (will be without the lowest score)
4. *Calculate the final score*

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Topic 5

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Discovering Algorithms by Manipulating Physical Objects

What if you come across a problem
for which you cannot find an algorithm you know
and you cannot figure out how to adapt any algorithms?

you can use a technique called:

MANIPULATING PHYSICAL OBJECTS

better know as:

playing around with things.

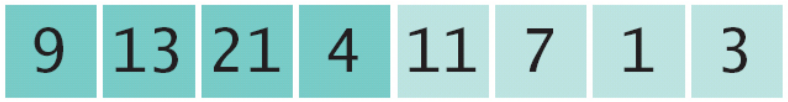
Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Manipulating Physical Objects: Example Problem

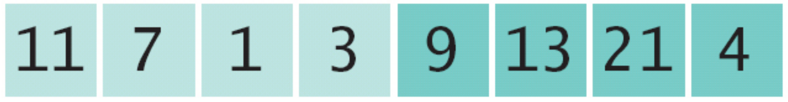
Here is a problem:

You are given an array whose size is an even number. You are to switch the first and the second half.

Before:



After:

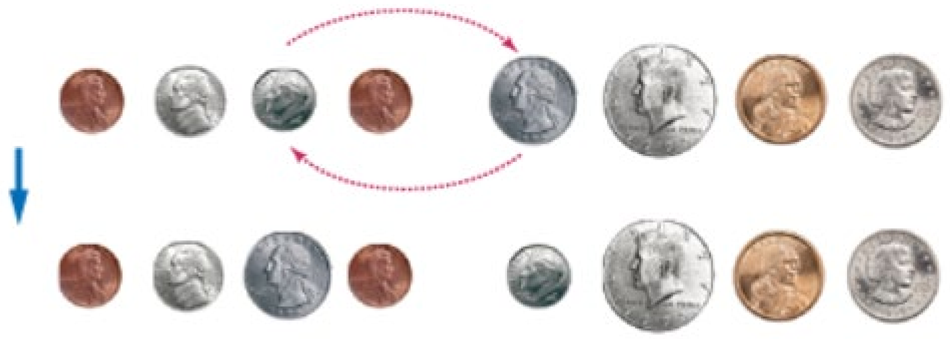


Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Manipulating Physical Objects: Coins

We'll use 8 coins as a model for our 8-elements of the array

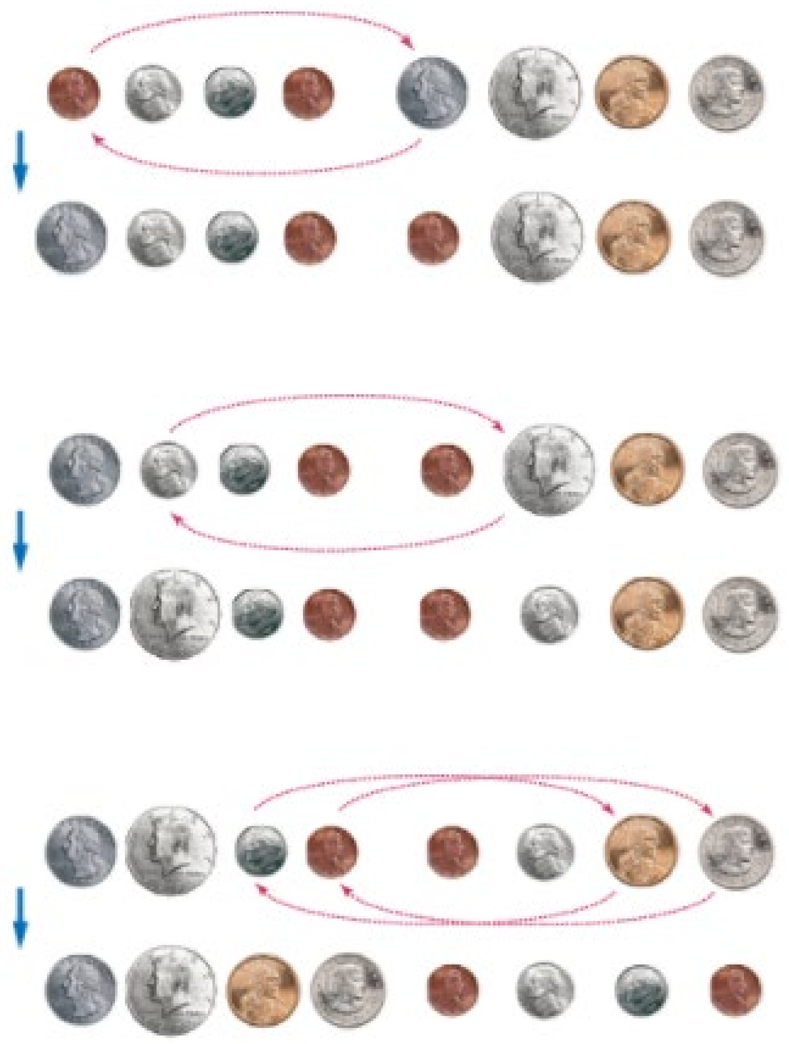
We can swap coins like we'd swap array elements:



Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Swapping Coins: the Algorithm

- We find that by swapping the
  - 1st and 4th coins, and
  - 2nd and 5th
  - 3rd and 6th
  - And 4th and 8th
  - We have swapped the first half of the 8 with the last



Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Translating the Manipulations to Code

Pseudocode:

```
i = 0
j = size / 2
While i < size / 2
    Swap elements at positions i and j.
    i++
    j++
```

Translating to C++ is left as a Programming Exercise at the end of the chapter

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Self Check: Practice Manipulating Objects

Using physical objects such as coins to represent array elements, determine the purpose of the function below:

```
void transform(int array[], int length)
{
   int position = 0;
   for (int k = 1; k < length; k++)
   {
      if (array[k] < array[position])
      {
         position = k;
      }
   }
   int temp = array[position];
   while (position > 0)
   {
      array[position] = array[position - 1];
      position--;
   }
   array[0] = temp;
}
```

ANSWER: copies the smallest value to the first array location and shifts other elements so no values are lost.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Topic 6

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Two-Dimensional Arrays

It often happens that you want to store collections of values that have a two-dimensional layout.

Such data sets commonly occur in financial and scientific applications.

An arrangement consisting of *tabular data (rows and columns* of values) is called:

a ***two-dimensional array***, or a ***matrix***

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Two-Dimensional Array Example

Consider the medal-count data from the 2014 Winter Olympic skating competitions:

Country	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Defining Two-Dimensional Arrays

C++ uses an array with *two* subscripts to store a *2D* array.

```
const int COUNTRIES = 8;
const int MEDALS = 3;
int counts[COUNTRIES][MEDALS];
```

An array with 8 rows and 3 columns is suitable for storing our medal count data.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Defining Two-Dimensional Arrays – Initializing

Just as with one-dimensional arrays, you *cannot* change the size of a two-dimensional array once it has been defined.

But you can initialize a 2-D array:

```
int counts[COUNTRIES][MEDALS] =
 {
    { 0, 3, 0 },
    { 0, 0, 1 },
    { 0, 0, 1 },
    { 1, 0, 0 },
    { 0, 0, 1 },
    { 3, 1, 1 },
    { 0, 1, 0 }
    { 1, 0, 1 }
 };
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Self Check: Declaring 2D Arrays

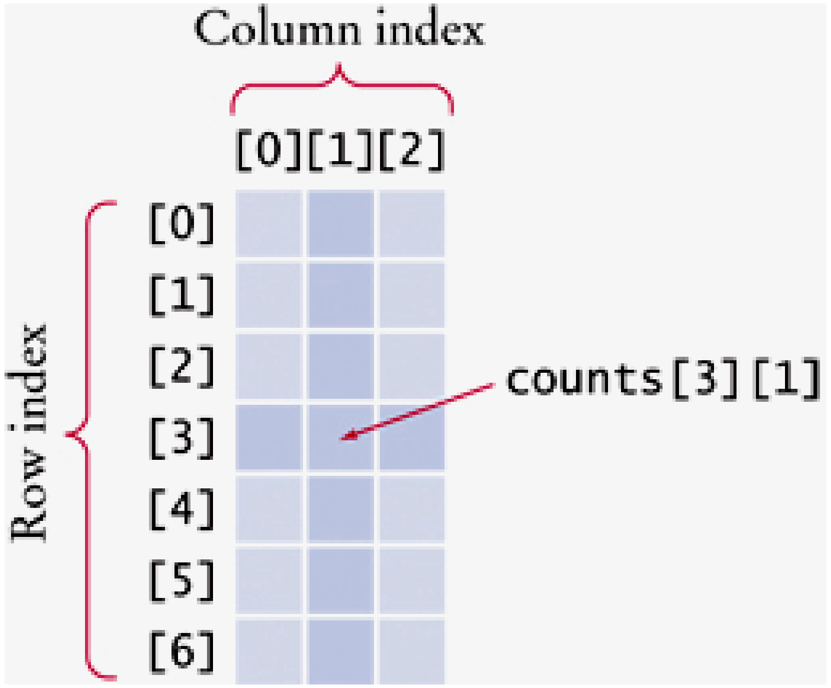
Write the code statement to declare an array `"a"` of integers with the specified properties.

Initialize elements to zero unless otherwise specified.

1. With 3 rows and 2 columns:
2. With 2 rows and 3 columns:
3. With 2 rows and 2 columns, containing 1 when the row and column index are the same:

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Two-Dimensional Arrays – Accessing Elements



```
// copy to value what is currently
// stored in the array at [3][1]
int value = counts[3][1];

// Then set that position in the array to 8
counts[3][1] = 8;
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Nested Loop to Print All Elements in a 2D Array

```
for (int i = 0; i < COUNTRIES; i++)
{
   // Process the ith row
   for (int j = 0; j < MEDALS; j++)
   {
      // Process the jth column in the ith row
      cout << setw(8) << counts[i][j];
   }
   // Start a new line at the end of the row
   cout << endl;
}
```

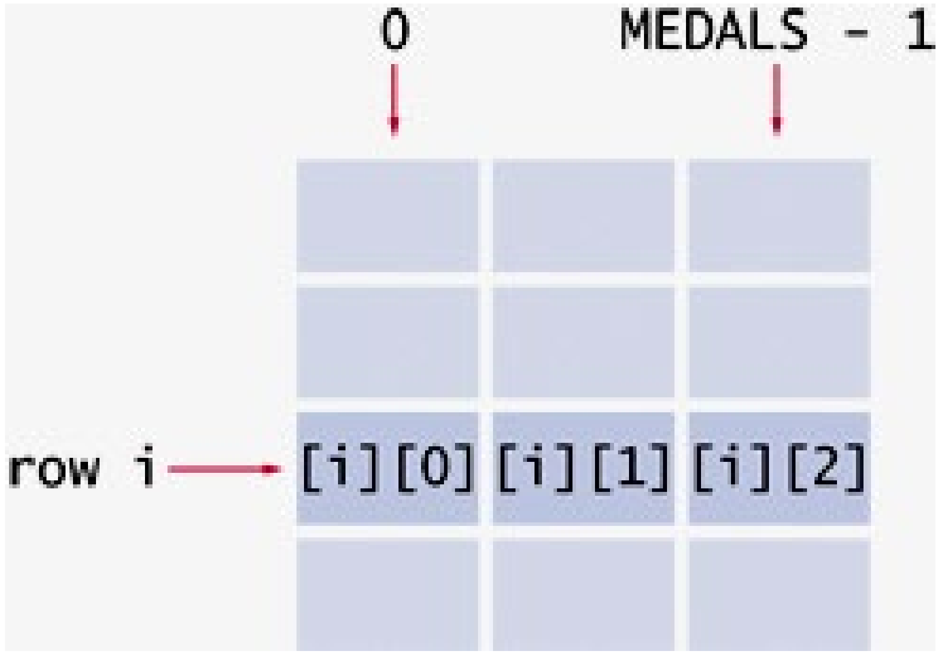
Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Computing Row and Column Totals

We must be careful to get the right indices.

For each row **`i`**, we must use the column indices:

`0, 1, … (MEDALS -1)`



Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Computing Row and Column Totals: Code Example

Column totals:

Let `j` be the silver column:

```
int total = 0; //loop to sum down the rows
for (int i = 0; i < COUNTRIES; i++)
{
   total = total + counts[i][j];
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Two-Dimensional Array Parameters

When passing a two-dimensional array to a function,

you must specify the number of columns *as a constant* when you write the parameter type, so the compiler can pre-calculate the memory addresses of individual elements.

This function computes the total of a given row.

```
const int COLUMNS = 3;
int row_total(int table[][COLUMNS], int row)
{
   int total = 0;
   for (int j = 0; j < COLUMNS; j++)
   {
      total = total + table[row][j];
   }
   return total;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Two-Dimensional Array Parameter Columns Hardwired

That function works for only arrays of 3 columns.

If you need to process an array
with a different number of columns, like 4,

you would have to write
a different function
that has 4 as the parameter.

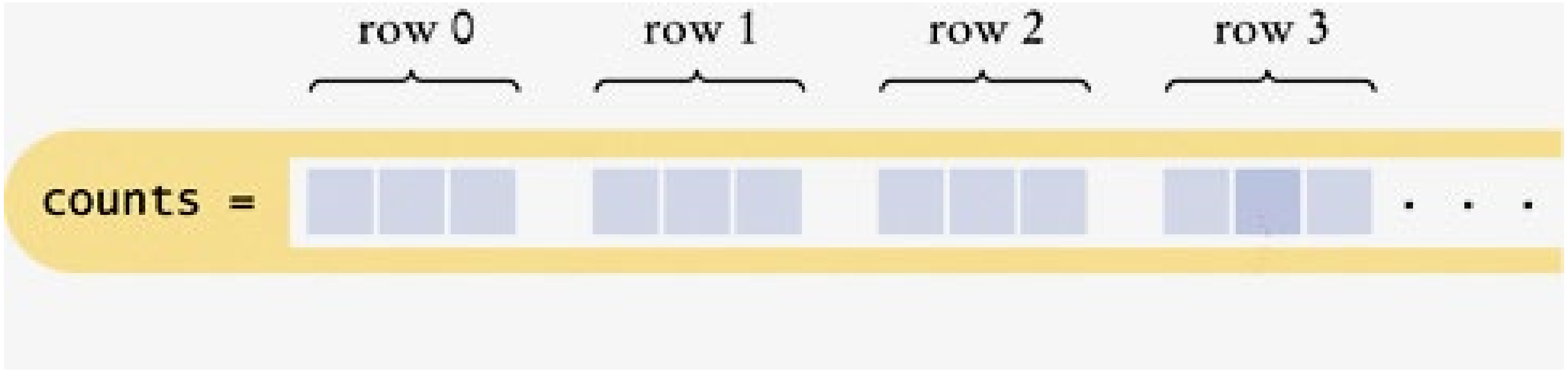
Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Two-Dimensional Array Storage

What's the reason behind this?

Although the array appears to be two-dimensional,
the elements are still stored as a linear sequence.

`counts` is stored as a sequence of rows, each 3 long.
So where is **`counts[3][1]`**?
The offset (calculated by the compiler) from the start of the array is
`3 x number of columns + 1`



Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Two-Dimensional Array Parameters: Rows

The **`row_total`** function did not need to know the number of rows of the array.

If the number of rows is required, pass it in:

```
int column_total(int table[][COLUMNS], int rows, int col)
{
   int total = 0;
   for (int i = 0; i < rows; i++)
   {
      total = total + table[i][col];
   }
   return total;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Two-Dimensional Array Parameters: Complete Code (1)

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

const int COLUMNS = 3;
/**
   Computes the total of a row in a table.
   @param table a table with 3 columns
   @param row the row that needs to be totaled
   @return the sum of all elements in the given row
*/
double row_total(int table[][COLUMNS], int row)
{
   int total = 0;
   for (int j = 0; j < COLUMNS; j++)
   {
      total = total + table[row][j];
   }
   return total;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Two-Dimensional Array Parameters: Complete Code (2)

```
int main()
{
    const int COUNTRIES = 8;
    const int MEDALS = 3;

    string countries[] =
    {"Canada","Italy","Germany","Japan", "Kazakhstan",
   "Russia", "South Korea", "United States"};

   int counts[COUNTRIES][MEDALS] =
    {
        { 0, 3, 0 },
        { 0, 0, 1 },
        { 0, 0, 1 },
        { 1, 0, 0 },
        { 0, 0, 1 },
        { 3, 1, 1 },
        { 0, 1, 0 }
        { 1, 0, 1 }
    };
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Two-Dimensional Array Parameters: Complete Code (3)

```
   cout << "    Country   Gold   Silver   Bronze    Total"
       << endl;

   // Print countries, counts, and row totals
   for (int i = 0; i < COUNTRIES; i++)
   {
      cout << setw(15) << countries[i];
      // Process the ith row
      for (int j = 0; j < MEDALS; j++)
      {
         cout << setw(8) << counts[i][j];
      }
      int total = row_total(counts, i);
      cout << setw(8) << total << endl;
   }
   return 0;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Practice It: 2D Array Parameters

Insert the missing statement. The function should return the result of adding the values in the first row of the 2D array received as argument.

```
int add_first_row(int array[][MAX_COLS], int rows,
int cols)
{
   int sum = 0;
   for (int k = 0; k < cols; k++)
   {
      sum = sum + ____________________;
   }
   return sum;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Arrays – Fixed Size is a Drawback

The size of an array *cannot* be changed after it is created.

You have to get the size right – *before* you define an array.

The compiler has to know the size to build it.
and a function must be told about the number
elements and possibly the capacity.

It cannot hold more than it’s initial capacity.

Next we'll discuss vectors, which have variable size and some other programmer-friendly features lacking in arrays.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Topic 7

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Vectors

The `vector` type was added to C++ to improve upon the array. Like the `string` type, it combines built-in member functions with data.

A **`vector`**

is not fixed in size when it is created

AND

you can keep putting things into it

forever!

(Until your computer runs out of RAM.)

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Declaring Vectors

When you declare a vector, you specify the type of the elements like you would with an array, but the type must be preceded by the word `vector`.

```
vector<double> data;
```

The element type must be in angle brackets. Other examples:

```
vector<int> counts;
vector<string> team_names;
```

By default, a vector is empty when created.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Declaring an non-empty Vector

You can specify the initial size.

You still must specify the type of the elements.

For example, here is a definition of a vector of **`double`**s whose initial size is **`10`**.

```
vector<double> data(10);
```

This is very close to the **`data`** *array* we used earlier.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Vector Examples: Table 2

`vector<int> numbers(10);`	A vector of ten integers.
`vector<string> names(3);`	A vector of three strings.
`vector<double> values;`	A vector of size 0.
`vector<double> values();`	**Error:** Does not define a vector.

`vector<int> numbers;`
`for (int i = 1; i <=10; i++)`
`{ numbers.push_back(i);}`
	A vector of ten integers, filled with 1, 2, 3, ..., 10.

`vector<int> numbers(10);`
`for (int i = 0; i < numbers.size(); i++)`
`{ numbers[i] = i + 1;}`
	Another way of defining a vector of ten integers 1, 2, 3, ..., 10.

`vector<int> numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };`	This syntax is supported with C++ 11 and above.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Accessing Elements in Vectors

You can access the elements in a vector
the same way as in an array, using an [index].

```
vector<double> values(4);
//display the forth element
cout << values[3] << end;
```

HOWEVER…

It is an error to access a element that is not in a vector.

```
vector<double> values(4);
//display the fifth element
cout << values[4] << end;   //ERROR
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vector push_back` and `pop_back`

The function ***push_back*** puts a value into a vector:

```
values.push_back(32); //32 added to end of vector
```

The `vector` increases its `size` by 1.

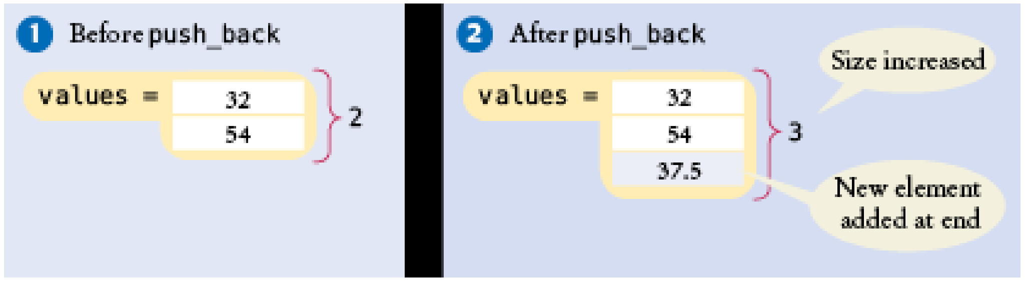
pop_back removes the last value placed into the vector, and the size decreases by 1:

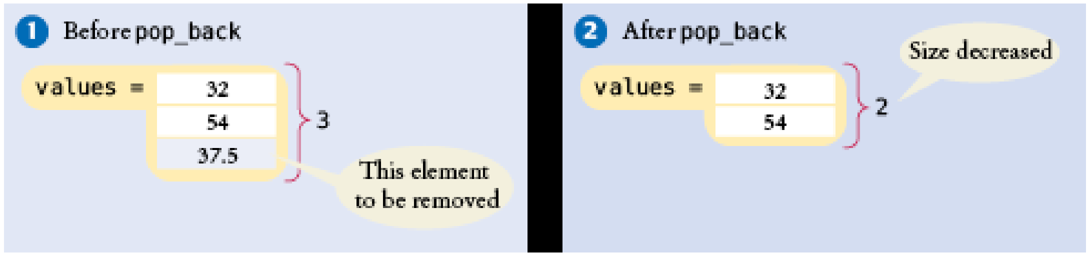
```
values.pop_back();
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`push_back` Adds Elements and Increments `size`

```
// an empty vector
vector<double> values;

values.push_back(32)
// values.size() now is 1

values.push_back(54);

values.push_back(37.5);
// values.size() now is 3

values.pop_back();
   //removes the 37.5
   //values.size()==2
```





Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`push_back` with User Input

You can use **`push_back`** to put user input into a vector:

```
vector<double> values;

double input;
while (cin >> input)
{
   values.push_back(input);
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

A Weakness of Arrays

With arrays, we must separately keep track of the current_size and the capacity. To display every element, we'd have to know the current size, assumed 10 below:

```
for (int i = 0; i < 10; i++)
{
   cout << values[i] << endl;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

vector size()

Vectors have the **size** member function which returns the current size of a vector.

The **vector** always knows how many elements are in it and you can always ask it to give you that quantity by calling the **size** method:

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vector` Parameters to Functions

The following function computes the sum of a `vector` of floating-point numbers:

```
double sum(vector<double> values)
{
   double total = 0;
   for (int i = 0; i < values.size(); i++)
   {
      total = total + values[i];
   }
   return total;
}
```

This function *visits* the `vector` elements, but does *not change* them.

Unlike an array, a `vector` is *passed by value* (copied) to a function, not passed by reference.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vector` Parameters – Changing the Values with &

If the function *should change* the values stored in the `vector`, then a `vector` reference must be passed, just like with `int` and `double` reference parameters. The `&` goes after the `<type>`:

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

For Efficiency, a Constant `vector` Reference

- Using a constant reference (Special Topic 5.2) for `vector` parameters avoids the need for the compiled code to copy the `vector` to feed the function, which could be inefficient
- Works only for parameters the function does not want to modify:

```
double sum2(const vector<double>& values)
{ … }
// const & added for efficiency
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vectors` Returned from Functions

Sometimes the function should ***`return`*** a **`vector`**.

Vectors are no different from any other data types in this regard.

Simply declare and build up the result in the function and return it:

```
vector<int> squares(int n)
{
    vector<int> result;
    for (int i = 0; i < n; i++)
    {
        result.push_back(i * i);
    }
    return result;
}
```

The function returns the squares from 0^2 up to $(n - 1)^2$ as a `vector`.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vector` Algorithms (Copying): Vectors Can Be Assigned!

Vectors do not suffer the limitations of arrays, where we had to include a code loop to copy each element.

```
vector<int> squares;

for (int i = 0; i < 5; i++)
{
   squares.push_back(i * i);
}

vector<int> lucky_numbers;
            // Initially empty

lucky_numbers = squares; //vector copy

         // Now lucky_numbers contains
         // the same elements as squares
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vector` Algorithms – Finding Matches

Suppose we want all the values in a vector that are greater than a certain value, say 100, in a vector.

Store them in another vector:

```
vector<double> matches;
const double MATCH=100;
for (int i = 0; i < values.size(); i++)
{
   if (values[i] > MATCH)
   {
      matches.push_back(values[i]);
   }
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vector` Algorithms – Removing an Element, Unordered

If you know the position of an element you want to remove from a `vector` in which the elements are not in any order, as you did in an array,

1. overwrite the element at that position with the last element
2. remove the last element with `pop_back()`

which also makes the vector smaller.

```
int last_pos = values.size() - 1;
   // Take the position of the last element
values[pos] = values[last_pos];
   // Replace element at pos with last element
values.pop_back();
   // Delete last element to make vector
   // one smaller
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vector` Algorithms – Removing an Element, Ordered

If you know the position of an element you want to remove from a `vector` in which the elements *are* in some order…

As you did in an array,

move all the elements after that position,

then remove the last element to reduce the size.

```
for (int i = pos + 1; i < values.size(); i++)
{
   values[i - 1] = values[i];
}
data.pop_back();
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vector` Algorithms – Inserting an Element, Unordered

When you need to insert an element into a `vector` whose elements are not in any order…

```
values.push_back(new_element);
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vector` Algorithms – Inserting an Element, Ordered

However when the elements in a `vector` are ordered, it's a bit more complicated, like it was in the array.

If you know the position, say **`pos`**, to insert the new element,

as in the array version, you need to move all the elements "up", but FIRST YOU GROW the `vector` by 1 to make room:

```
int last_pos = values.size() - 1;
values.push_back(values[last_pos]); //grow it
for (int i = last_pos; i > pos; i--)
{
   values[i] = values[i - 1];
}
values[pos] = new_element;
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vector` Algorithms: Sorting with the C++ Library

Recall that you call the **`sort`** function to sort an array.

This can be used on vectors also.

The syntax for `vector`s uses 2 more built-in `vector` functions, which tell the address (a reference) to the first and last `vector` elements:

```
sort(values.begin(), values.end());
```

Don't forget to

```
#include <algorithm>
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Two Dimensional `vectors`: a `vector` of `vectors`

There are no 2D `vector`s, but if you want to store rows and columns, you can use a `vector` of `vector`s. For example, the medal counts of Section 6.6:

```
vector<vector<int>> counts;
//counts is a vector of rows. Each row is a vector<int>
```

You need to initialize it, to make sure there are rows and columns for all the elements.

```
for (int i = 0; i < COUNTRIES; i++)
{
   vector<int> row(MEDALS);
   counts.push_back(row);
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

vector of vectors

- You can access the `vector counts[i][j]` in the same way as 2D arrays.
  - `counts[i]` denotes the ith row, and `counts[i][j]` is the value in the jth column of that row.

- The advantage over 2D arrays:
  - `vector` row and column sizes don't have to be fixed at compile time.

```
int countries = . . .;
int medals = . . .;
vector<vector<int>> counts;
for (int i = 0; i < countries; i++)
{
   vector<int> row(medals);
   counts.push_back(row);
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

`vector` of `vectors`: Determining the row/column sizes

To find the number of rows and columns:

```
vector<vector<int>> values = . . .;

int rows = values.size();
int columns = values[0].size();
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Which to Use? `vector` or array?

- For most programming tasks, `vector`s are easier to use than arrays. A `vector`:
  - can grow and shrink.
  - remembers its size.
  - Has handy built-in functions like
    - **`begin(), end()`**
    - **`push_back(), pop_back()`**
    - **`size()`**
    - **`at():`** this is an alternative to the [] notation to choose an element, and includes bounds checking to detect invalid subscripts
- Advanced programmers may prefer arrays for efficiency.
- You still need to to use arrays if you work with older programs or use C without the "++", such as in microcontroller applications.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

The Range-Based `for` Loop

C++ 11 and after added a convenient `for()` syntax to visiting all elements in a “range” in a `vector`. No index variable nor comparison is needed:

```
vector<int> values = {1, 4, 9, 16, 25, 36};
for (int v : values) //visits all elements
{
   cout << v << " ";
}
```

If you want to modify elements, you must ***declare the loop variable as a reference:***

```
for (int& v : values) // & allows modifying the vector elements
{
   v++; //increment every element
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

The Range-Based `for` Loop also Works for Arrays

The range-based `for` loop also works for arrays. For example:

```
int primes[] = { 2, 3, 5, 7, 11, 13 };
for (int p : primes)
{
   cout << p << " ";
}
```

However, the range based `for` will loop over the entire capacity of the array, whether it is filled or partially empty.

Finally, you can use `auto` instead of the element type, for either arrays or vectors:

```
for (auto p : primes)
{
   cout << p << " ";
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

CHAPTER SUMMARY #1

Use arrays for collecting values.

- •Use an array to collect a sequence of values of the same type.
- •Individual elements in an array values are accessed by an integer index i, using the notation `values[i]`.
- •An array element can be used like any variable.
- •An array index must be at least zero and less than the size of the array.
- •A bounds error, which occurs if you supply an invalid array index, can corrupt data or cause your program to terminate.
- •With a partially filled array, keep a companion variable for the current size.

Be able to use common array algorithms.

- •To copy an array, use a loop to copy its elements to a new array.
- •When separating elements, don't place a separator before the first element.
- •A linear search inspects elements in sequence until a match is found.
- •Before inserting an element, move elements to the end of the array starting with the last one.
- •Use a temporary variable when swapping two elements.

Copyright © 2018 by John Wiley & Sons. All rights reserved

CHAPTER SUMMARY #2

Implement functions that process arrays.

- When passing an array to a function, also pass the size of the array.
- Array parameters are always reference parameters.
- A function's return type cannot be an array.
- When a function modifies the size of an array, it needs to tell its caller.
- A function that adds elements to an array needs to know its capacity.

Be able to combine and adapt algorithms for solving a programming problem.

- By combining fundamental algorithms, you can solve complex programming tasks.
- Be familiar with the implementation of fundamental algorithms so that you can adapt them.

Discover algorithms by manipulating physical objects.

- Use a sequence of coins, playing cards, or toys to visualize an array of values.
- You can use paper clips as position markers or counters.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

CHAPTER SUMMARY #3

Use two-dimensional arrays for data that is arranged in rows and columns.

- Use a two-dimensional array to store tabular data.
- Individual elements in a two-dimensional array are accessed by using two subscripts, `array[i][j]`.
- A two-dimensional array parameter must have a fixed number of columns.

Use vectors for managing collections whose size can change.

- A `vector` stores a sequence of values whose size can change.
- Use the `size` member function to obtain the current size of a `vector`.
- Use the `push_back` member function to add more elements to a `vector`. Use `pop_back` to reduce the size.
- `vector`s can occur as function arguments and return values.
- Use a reference parameter (`vector<int>&`) to modify the contents of a `vector`.
- A function can return a `vector`.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved