Topic 7

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

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

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

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.

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
```


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.

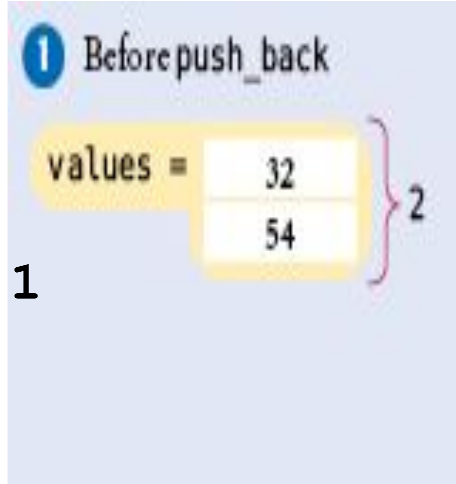
pop_back removes the last value placed into the vector, and the size decreases by 1:

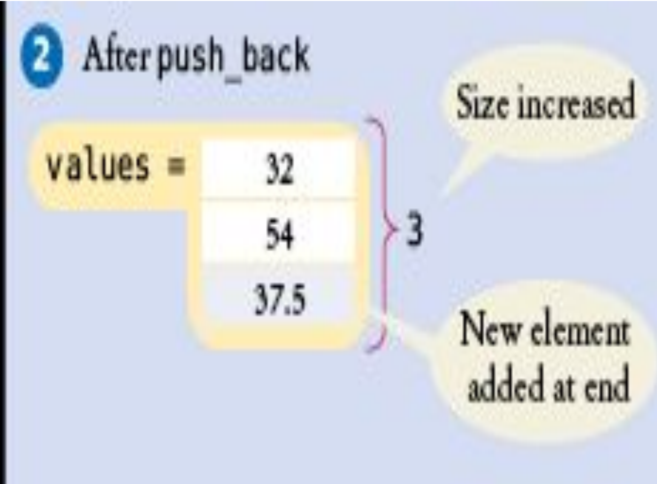
```
values.pop_back();
```

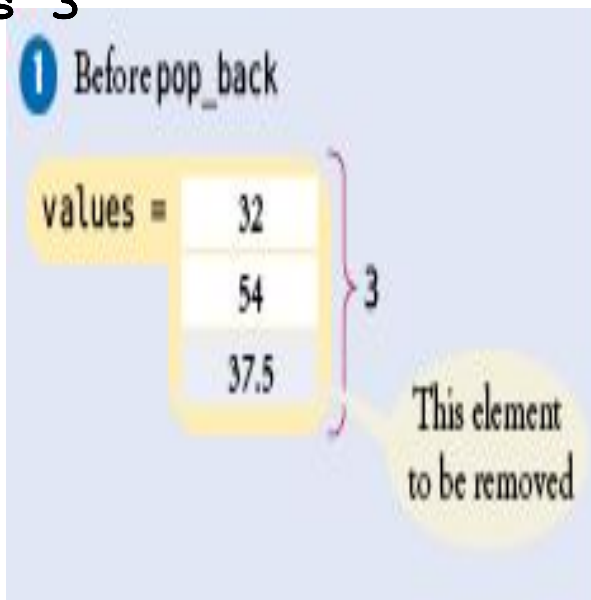
Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

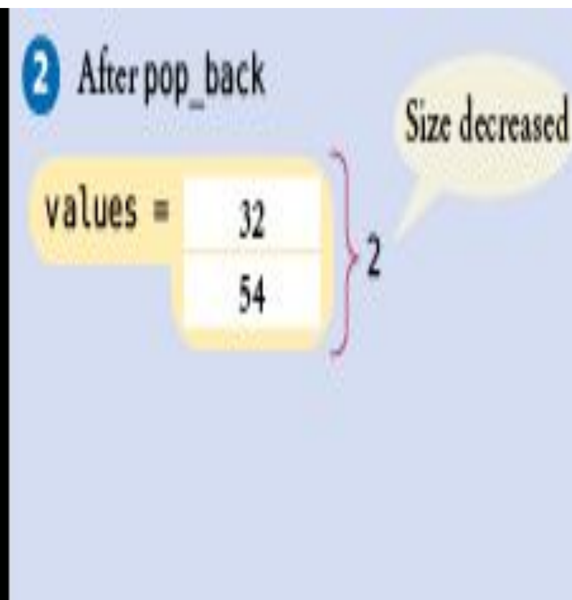
push_back Adds Elements and Increments size

```
// 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
```









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.

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;
   }
}
```


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
```


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;
```


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);
}
```


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();
```


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.

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 << " ";
}
```


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

CHAPTER SUMMARY #3

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`.

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