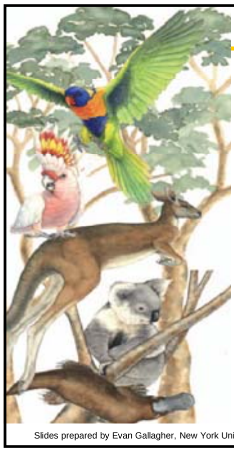


Chapter Six: Arrays and Vectors

Slides prepared by Evan Gallagher, New York University

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 learn how to use two-dimensional arrays

Using Arrays and Vectors

Mail, mail and more mail – how to manage it?

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

Using Arrays

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

Using Arrays and Vectors

In both vectors and arrays,
the stored data is of
the *same* type

Using Arrays and Vectors

Think of a sequence of data:

32 54 67.5 29 35 80 115 44.5 100 65

(all of the same type – real numbers)

Using Arrays and Vectors

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.)

Using Arrays and Vectors

32 54 67.5 29 35 80 115 44.5 100 65

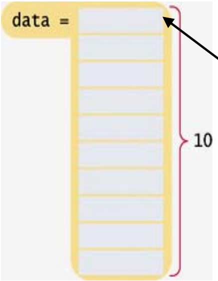
So you would create a variable for each,
of course!

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

Then what ???

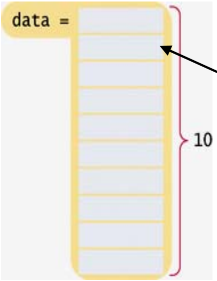
Using Arrays and Vectors

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



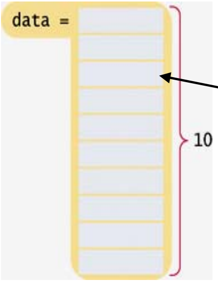
Hm. What is the the max, so far?

Using Arrays and Vectors



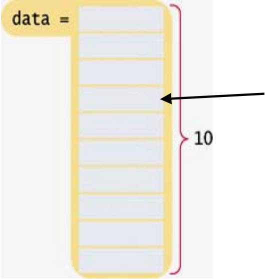
Maybe this one?

Using Arrays and Vectors



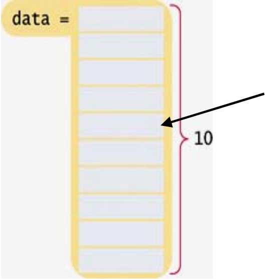
Or this one?

Using Arrays and Vectors



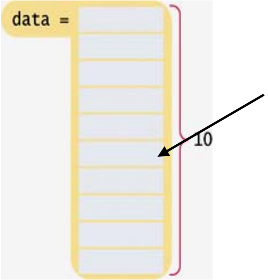
Or this one?

Using Arrays and Vectors



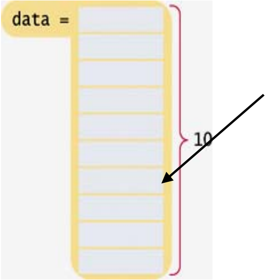
How about here?

Using Arrays and Vectors



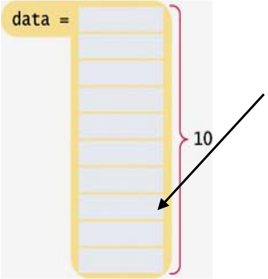
Gotta check here too!

Using Arrays and Vectors



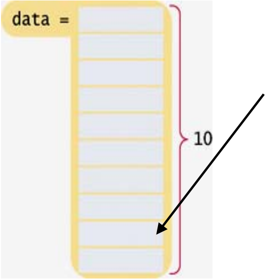
Again, maybe this one?

Using Arrays and Vectors



Or this one?

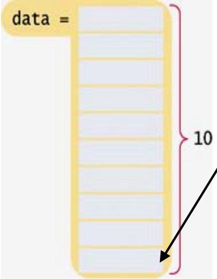
Using Arrays and Vectors



Or this one?

Will this never end!

Using Arrays and Vectors



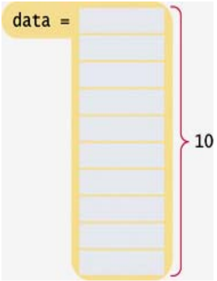
Or the last one? ***Finally!***

Using Arrays and Vectors

That would have been impossible with ten separate variables!

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

Defining Arrays



An “array of double type”

Ten elements of **`double`** type can be stored under one name as an array.

```
double data[10];
```

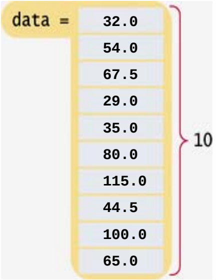
type of each element

quantity of elements – the “size” of the array

Defining Arrays with Initialization

When you define an array, you can specify the initial values:

```
double data[] = { 32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65 };
```



Accessing an Array Element

An array element can be used like any variable.

To access an array element, you use the notation:

```
data[i]
```

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

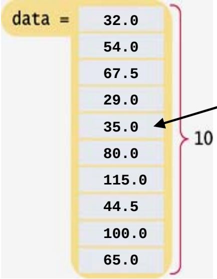
Accessing an Array Element

Put the junk mail in there

in `mailboxes[356]`

Accessing an Array Element

To access the element at index 4 using this notation: data[4]
4 is the *index*.



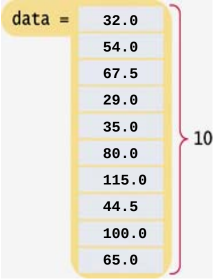
```
cout << data[4] << endl;
```

The output will be **`35.0`**.

C++ for Everyone by Cay Horstmann

Accessing an Array Element

The same notation can be used to change the element.

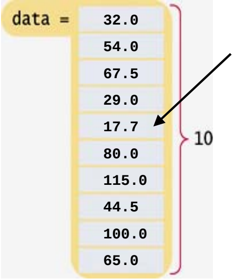


```
data[4] = 17.7;
```

C++ for Everyone by Cay Horstmann

Accessing an Array Element

The same notation can be used to change the element.

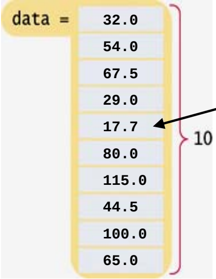


```
data[4] = 17.7;
```

C++ for Everyone by Cay Horstmann

Accessing an Array Element

The same notation can be used to change the element.



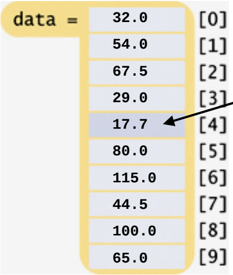
```
data[4] = 17.7;
cout << data[4] << endl;
```

The output will be **`17.7`**.

C++ for Everyone by Cay Horstmann

Accessing an Array Element

You might have thought those last two slides were wrong:
`data[4]` is getting the data from the “fifth” element.



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

In C++ and most computer languages, indexing starts with **`0`**.

C++ for Everyone by Cay Horstmann

Accessing an Array Element

That is, the legal elements for the **`data`** array are:

`data[0]`, the ***first*** element
`data[1]`, the second element
`data[2]`, the third element
`data[3]`, the fourth element
`data[4]`, the fifth element
`...`
`data[9]`, the tenth ***and last legal*** element

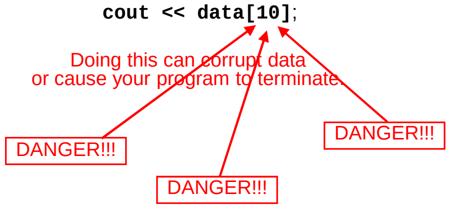
The index must be **`>= 0`** and **`<= 9`**, the size.

C++ for Everyone by Cay Horstmann

Illegally Accessing an Array Element – *Bounds Error*

A *bounds* error occurs when you access
an element outside the legal set of indices:

```
cout << data[10];
```



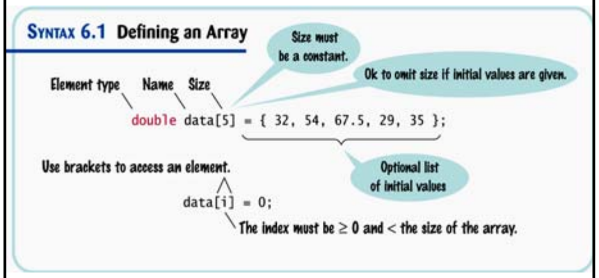
Doing this can corrupt data
or cause your program to terminate.

DANGER!!!

DANGER!!!

DANGER!!!

Array Syntax



Syntax 6.1 Defining an Array

```
double data[5] = { 32, 54, 67.5, 29, 35 };
```

Element type
Name
Size
Size must be a constant.
Ok to omit size if initial values are given.
Optional list of initial values

Use brackets to access an element.

```
data[i] = 0;
```

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

Array Syntax

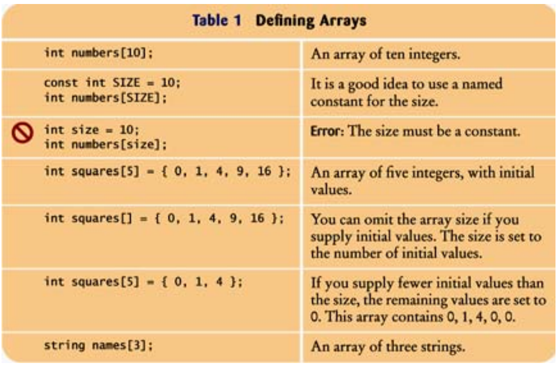


Table 1 Defining Arrays

`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];`	**Error:** The size must be a constant.
`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.

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A **`for`** loop's variable is best:

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

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A **`for`** loop's variable is best:

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

When **`i`** is **`0`**,

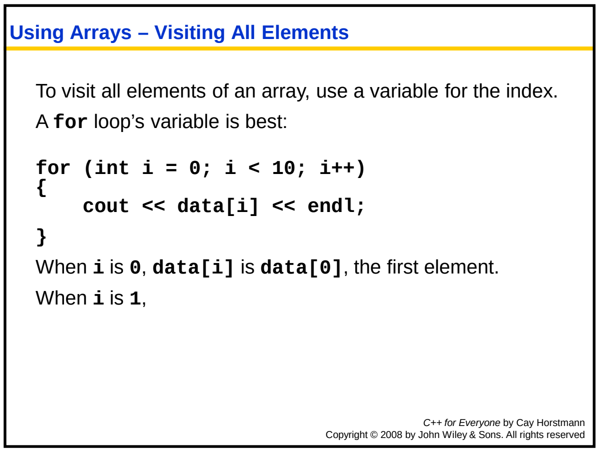
Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A **`for`** loop's variable is best:

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

When **`i`** is **`0`**, **`data[i]`** is **`data[0]`**, the first element.



Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

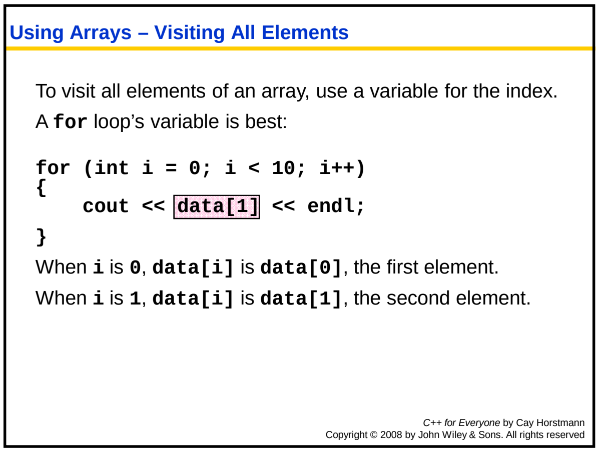
A `for` loop's variable is best:

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

When `i` is `0`, `data[i]` is `data[0]`, the first element.

When `i` is `1`,

C++ for Everyone by Cay Horstmann



Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

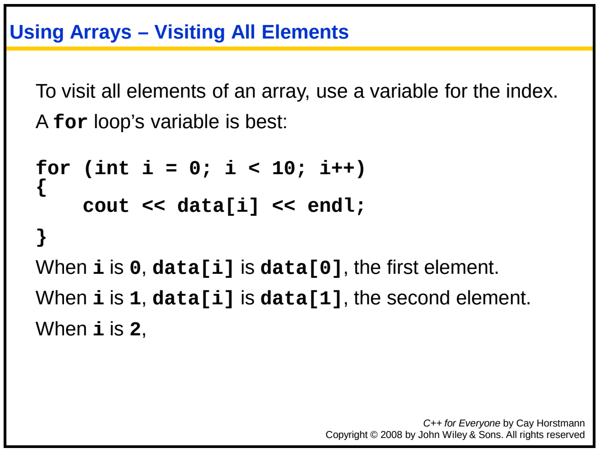
A `for` loop's variable is best:

```
for (int i = 0; i < 10; i++)
{
    cout << data[1] << endl;
}
```

When `i` is `0`, `data[i]` is `data[0]`, the first element.

When `i` is `1`, `data[i]` is `data[1]`, the second element.

C++ for Everyone by Cay Horstmann



Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A `for` loop's variable is best:

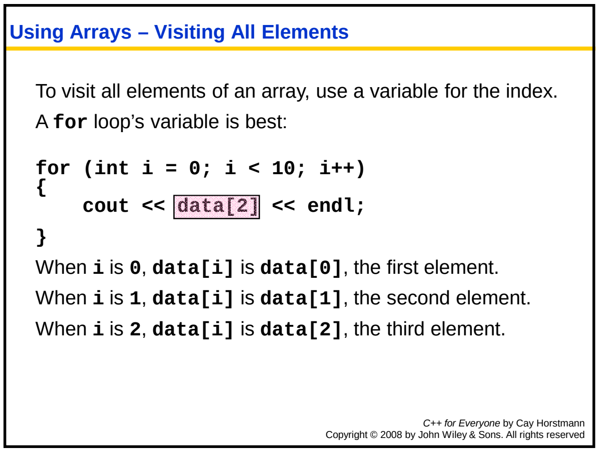
```
for (int i = 0; i < 10; i++)
{
    cout << data[i] << endl;
}
```

When `i` is `0`, `data[i]` is `data[0]`, the first element.

When `i` is `1`, `data[i]` is `data[1]`, the second element.

When `i` is `2`,

C++ for Everyone by Cay Horstmann



Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A `for` loop's variable is best:

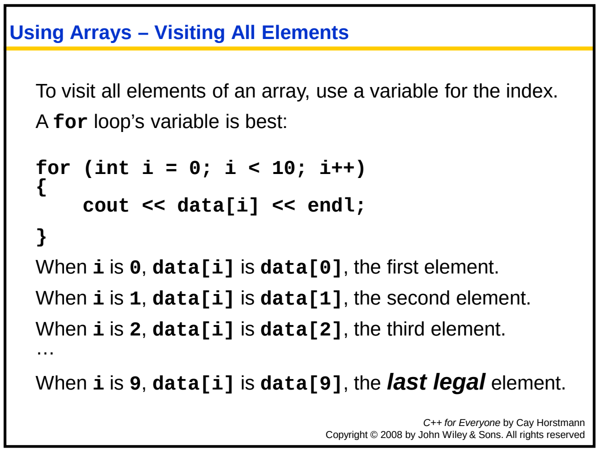
```
for (int i = 0; i < 10; i++)
{
    cout << data[2] << endl;
}
```

When `i` is `0`, `data[i]` is `data[0]`, the first element.

When `i` is `1`, `data[i]` is `data[1]`, the second element.

When `i` is `2`, `data[i]` is `data[2]`, the third element.

C++ for Everyone by Cay Horstmann



Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A `for` loop's variable is best:

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

When `i` is `0`, `data[i]` is `data[0]`, the first element.

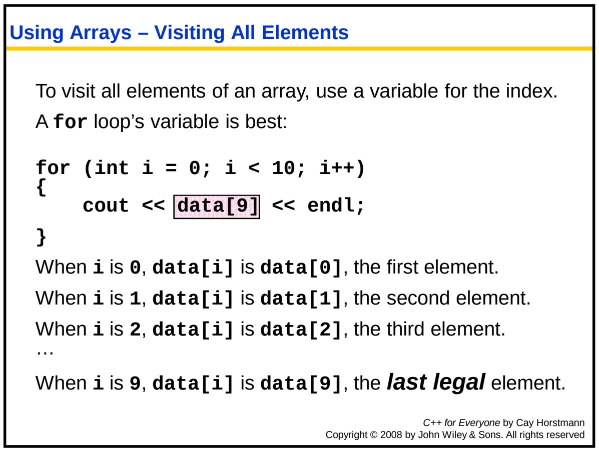
When `i` is `1`, `data[i]` is `data[1]`, the second element.

When `i` is `2`, `data[i]` is `data[2]`, the third element.

…

When `i` is `9`, `data[i]` is `data[9]`, the ***last legal*** element.

C++ for Everyone by Cay Horstmann



Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A `for` loop's variable is best:

```
for (int i = 0; i < 10; i++)
{
    cout << data[9] << endl;
}
```

When `i` is `0`, `data[i]` is `data[0]`, the first element.

When `i` is `1`, `data[i]` is `data[1]`, the second element.

When `i` is `2`, `data[i]` is `data[2]`, the third element.

…

When `i` is `9`, `data[i]` is `data[9]`, the ***last legal*** element.

C++ for Everyone by Cay Horstmann

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.

A **for** loop's variable is best:

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

Note that the loop condition is that the index is

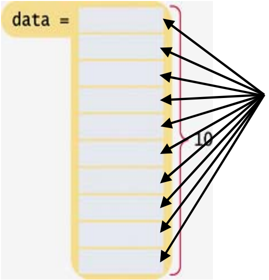
***less than* 10**

because there is no element corresponding to data[10].

But 10 ***is*** the number of elements we want to visit.

Using Arrays and Vectors

You would use a **for** statement to find
the maximum in an array.



for each element,
- check this one

Arrays – One 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.

What is the size?

That can be a hard question sometimes!

Vectors

Vectors to the rescue!

Vectors

A ***vector*** stores a sequence of values,

just like the array does,

but its size can change.

Defining Vectors

When you define a vector, you
must specify the type of the elements.

```
vector< T > data;
```

Defining Vectors

When you define a vector, you
must specify the type of the elements.

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

Note that the element type is enclosed in angle brackets.

`data` can contain **`double`**s

Defining Vectors

By default, a vector is empty when created.

```
vector<double> data; // data is empty
```

Defining Vectors

You can specify the initial size.

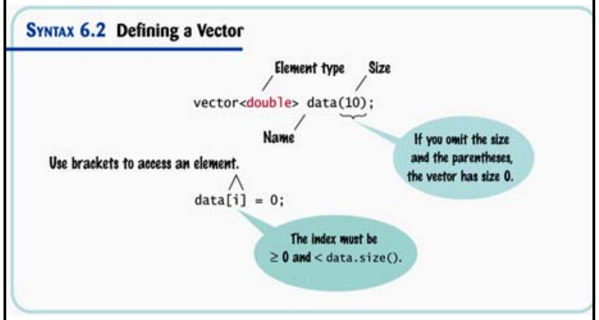
You still must specify the type of the elements.

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.

Defining Vectors



SYNTAX 6.2 Defining a Vector

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

Element type

Size

Name

If you omit the size and the parentheses, the vector has size 0.

Use brackets to access an element.

```
data[i] = 0;
```

The index must be ≥ 0 and < data.size().

Accessing Elements in Vectors

You access the elements in a vector
the same was as in an array, using an index.

```
vector<double> data(10);
//display the forth element
cout << data[3] << end;
```

HOWEVER...

Accessing Elements in Vectors

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

EMPTY!

```
vector<double> data;
//display the forth element
cout << data[3] << end;
```

ERROR!

push_back

So how do you put values into a vector?

You stuff them in—

— at the end!

pop_back

And how do you take them out?

You pop ‘em!

push_back and pop_back

The method ***push_back*** is used to put a value into a vector:

```
data.push_back( 32 );
```

adds the value **`32.0`** to the vector named **`data`**.

push_back and pop_back

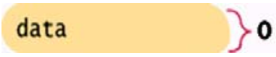
The method ***pop_back*** removes value into a vector

`pop_back` removes the last value placed
into the vector with **`push_back`**.

```
data.pop_back();
```

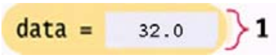
removes a value from the vector named **`data`**.

push_back Adds an Element



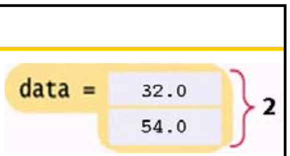
```
vector<double> data;
// Now data is empty
// size is 0
```

push_back Adds an Element



```
vector<double> data;

data.push_back(32);
// Now data has size 1
// and element 32
```

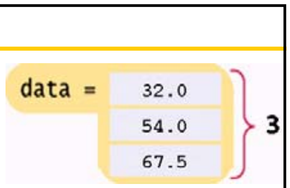
push_back Adds an Element



```
vector<double> data;

data.push_back(32);
data.push_back(54);
// Now data has size 2
// and elements 32, 54
```

C++ for Everyone by Cay Horstmann

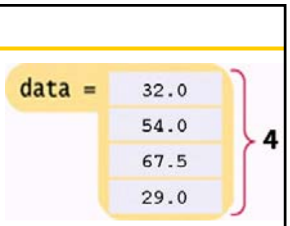
push_back Adds an Element



```
vector<double> data;

data.push_back(32);
data.push_back(54);
data.push_back(67.5);
// Now data has size 3
// and elements 32, 54, 67.5
```

C++ for Everyone by Cay Horstmann

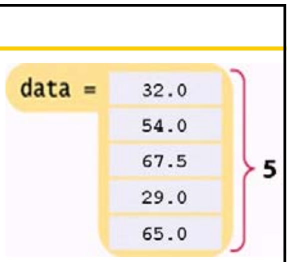
push_back Adds an Element



```
vector<double> data;

data.push_back(32);
data.push_back(54);
data.push_back(67.5);
data.push_back(29);
// Now data has size 4
// and elements 32, 54, 67.5, 29
```

C++ for Everyone by Cay Horstmann

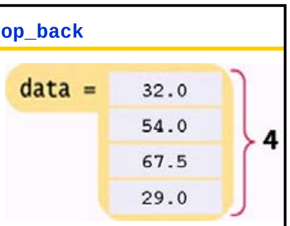
push_back Adds an Element



```
vector<double> data;

data.push_back(32);
data.push_back(54);
data.push_back(67.5);
data.push_back(29);
data.push_back(65);
// Now data has size 5
// and elements 32, 54, 67.5, 29, 65
```

C++ for Everyone by Cay Horstmann

Removing the Last Element with **pop_back**



```
vector<double> data;

data.push_back(32);
data.push_back(54);
data.push_back(67.5);
data.push_back(29);
data.push_back(65);
data.pop_back();
// Now data has size 4
// and elements 32, 54, 67.5, 29
```

C++ for Everyone by Cay Horstmann

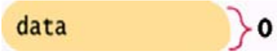
push_back and pop_back

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

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

C++ for Everyone by Cay Horstmann

push_back Adds an Element

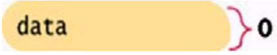


```
vector<double> data;

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

We are staring again with an empty vector

C++ for Everyone by Cay Horstmann

push_back Adds an Element

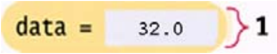


```
vector<double> data;

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

The user types **32**

C++ for Everyone by Cay Horstmann

push_back Adds an Element

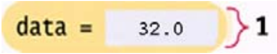


```
vector<double> data;

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

32 is placed into the vector

C++ for Everyone by Cay Horstmann

push_back Adds an Element

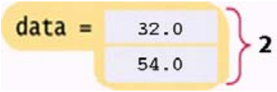


```
vector<double> data;

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

The user types **54**

C++ for Everyone by Cay Horstmann

push_back Adds an Element

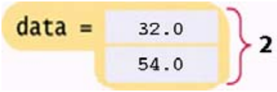


```
vector<double> data;

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

54 is placed into the vector

C++ for Everyone by Cay Horstmann

push_back Adds an Element

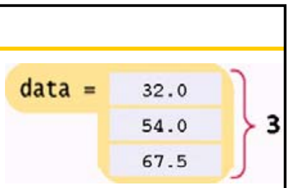


```
vector<double> data;

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

The user types `67.5`

C++ for Everyone by Cay Horstmann

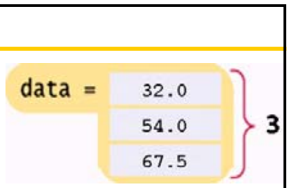
push_back Adds an Element



```
vector<double> data;

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

67.5 is placed into the vector

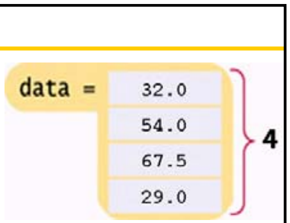
push_back Adds an Element



```
vector<double> data;

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

The user types **29**

push_back Adds an Element



```
vector<double> data;

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

29 is placed into the vector

Defining Vectors

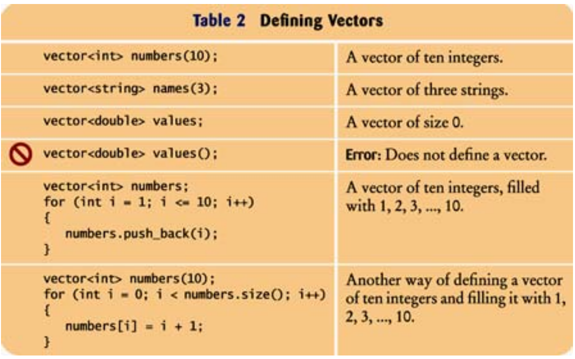


Table 2 Defining Vectors

`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 and filling it with 1, 2, 3, ..., 10.

Using Vectors – Visiting Every Element

How do you visit every element in an vector?

Using Vectors – Visiting Every Element

With arrays, to display every element, it would be:

?

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

But with vectors, we don't know about that 10!

Using Vectors – Visiting Every Element

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

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

Partially-Filled Arrays

Unlike a vector, an array cannot change size at run time.

There is no analog to the
push_back or **pop_back** member functions.

So it's the same question as before:

What is the size?

Partially-Filled Arrays – Capacity

What is the size?

We guess.

Well, we don't just guess – we read the problem
and try to pick a reasonable maximum number of elements

We call this quantity the *capacity*.

Partially-Filled Arrays – Capacity

For example, we may decide for a particular problem
that there at least ten values, but never more than 100.

We would set the capacity with a **const**:

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

Partially-Filled Arrays

This array will usually have less than CAPACITY elements in it

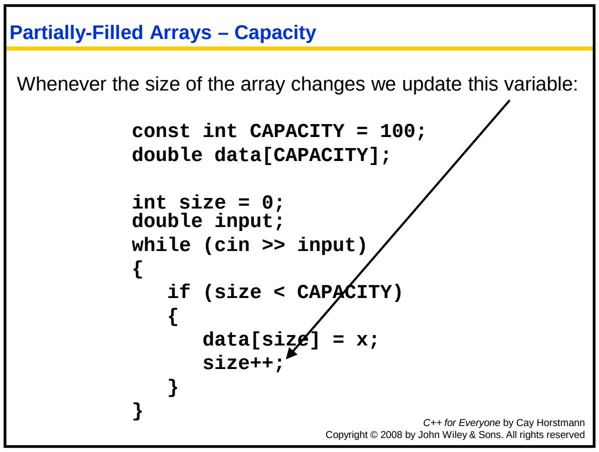
We call this kind of array a *partially filled array*.

Partially-Filled Arrays – Companion Variable for Size

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

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

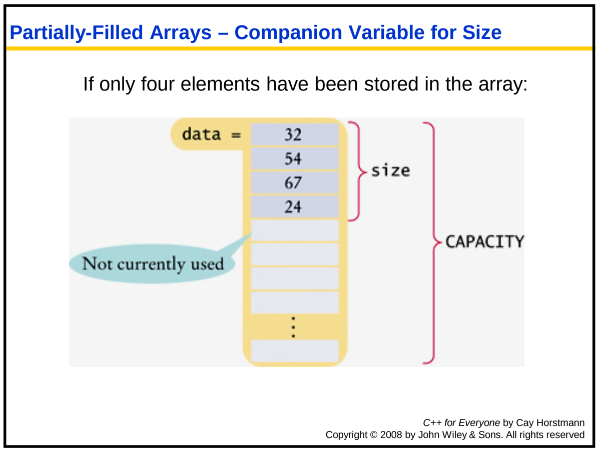
```
const int CAPACITY = 100;
double data[CAPACITY];

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



Partially-Filled Arrays – Capacity

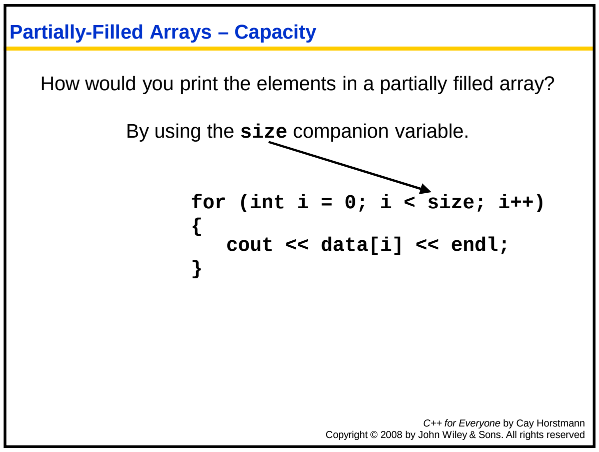
Whenever the size of the array changes we update this variable:

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

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



Partially-Filled Arrays – Companion Variable for Size

If only four elements have been stored in the array:

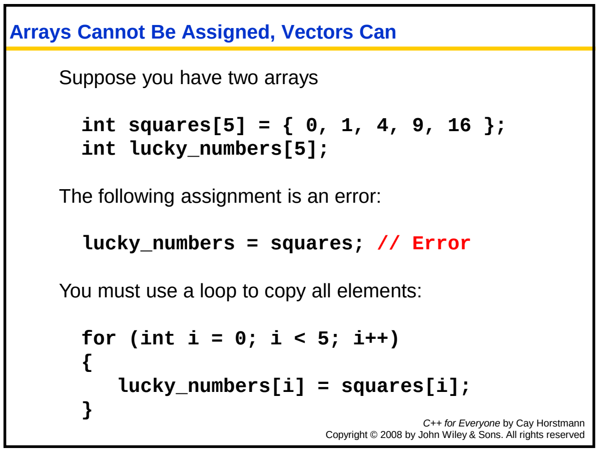


Partially-Filled Arrays – Capacity

How would you print the elements in a partially filled array?

By using the **`size`** companion variable.

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



Arrays Cannot Be Assigned, Vectors Can

Suppose you have two arrays

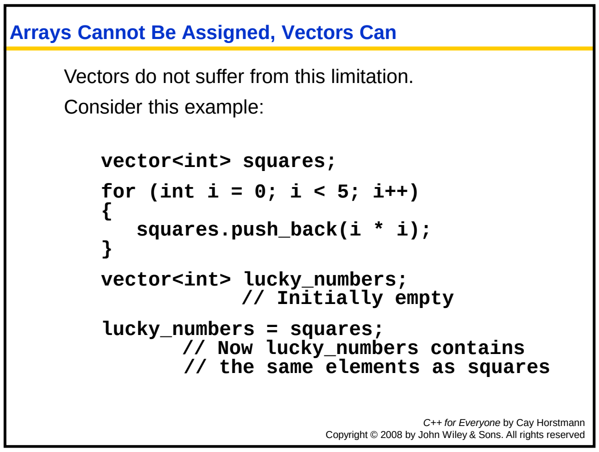
```
int squares[5] = { 0, 1, 4, 9, 16 };
int lucky_numbers[5];
```

The following assignment is an error:

```
lucky_numbers = squares; // Error
```

You must use a loop to copy all elements:

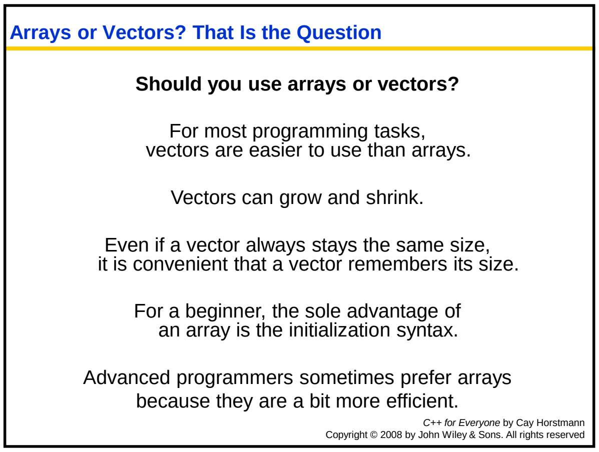
```
for (int i = 0; i < 5; i++)
{
   lucky_numbers[i] = squares[i];
}
```



Arrays Cannot Be Assigned, Vectors Can

Vectors do not suffer from this limitation.
Consider this example:

```
vector<int> squares;
for (int i = 0; i < 5; i++)
{
   squares.push_back(i * i);
}
vector<int> lucky_numbers;
            // Initially empty
lucky_numbers = squares;
       // Now lucky_numbers contains
       // the same elements as squares
```



Arrays or Vectors? That Is the Question

Should you use arrays or vectors?

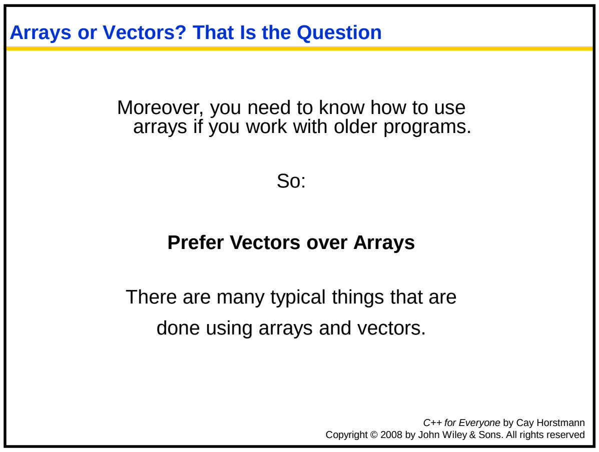
For most programming tasks,
vectors are easier to use than arrays.

Vectors can grow and shrink.

Even if a vector always stays the same size,
it is convenient that a vector remembers its size.

For a beginner, the sole advantage of
an array is the initialization syntax.

Advanced programmers sometimes prefer arrays
because they are a bit more efficient.



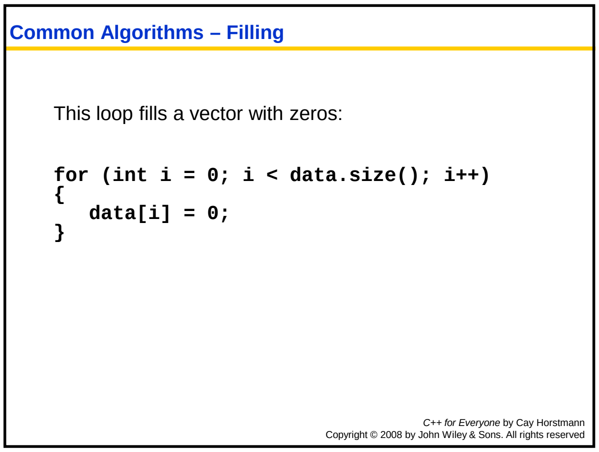
Arrays or Vectors? That Is the Question

Moreover, you need to know how to use arrays if you work with older programs.

So:

Prefer Vectors over Arrays

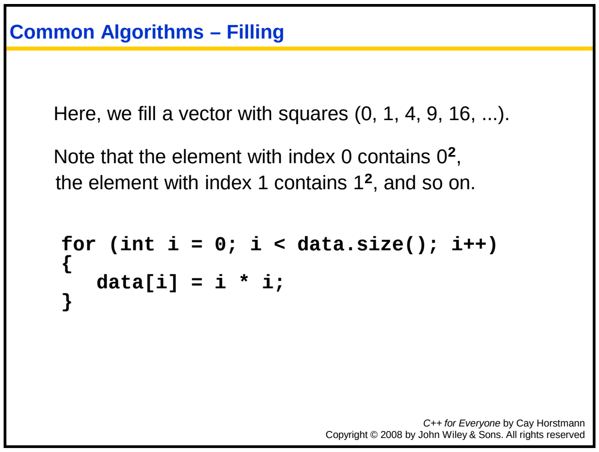
There are many typical things that are done using arrays and vectors.



Common Algorithms – Filling

This loop fills a vector with zeros:

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

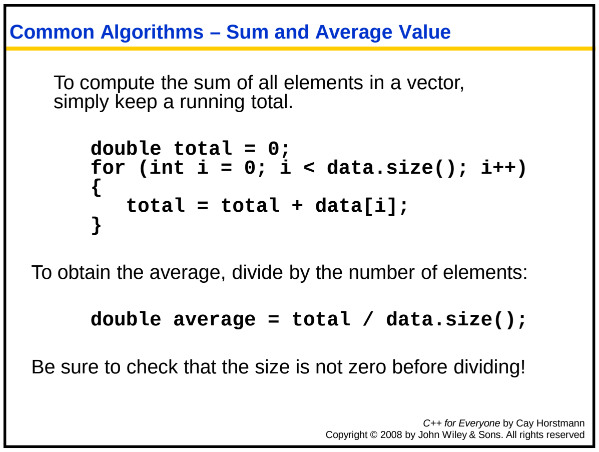


Common Algorithms – Filling

Here, we fill a vector with squares (0, 1, 4, 9, 16, ...).

Note that the element with index 0 contains 0^2, the element with index 1 contains 1^2, and so on.

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



Common Algorithms – Sum and Average Value

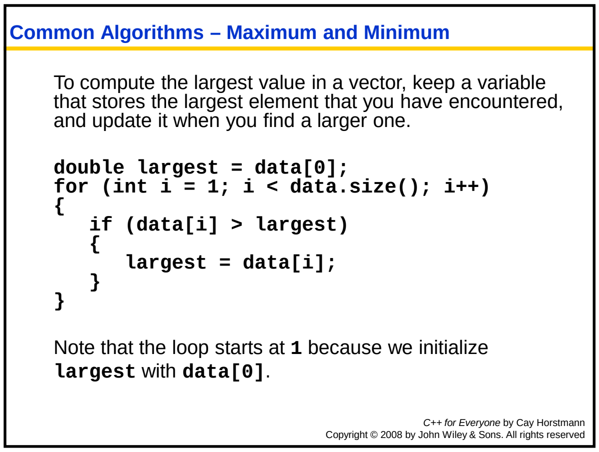
To compute the sum of all elements in a vector, simply keep a running total.

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

To obtain the average, divide by the number of elements:

```
double average = total / data.size();
```

Be sure to check that the size is not zero before dividing!

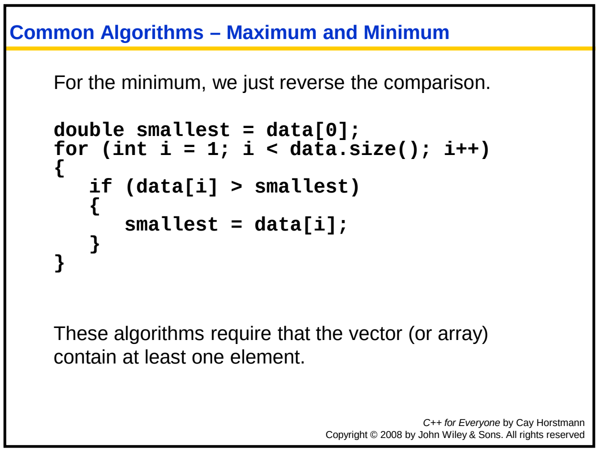


Common Algorithms – Maximum and Minimum

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 = data[0];
for (int i = 1; i < data.size(); i++)
{
   if (data[i] > largest)
   {
      largest = data[i];
   }
}
```

Note that the loop starts at `1` because we initialize `largest` with `data[0]`.



Common Algorithms – Maximum and Minimum

For the minimum, we just reverse the comparison.

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

These algorithms require that the vector (or array) contain at least one element.

Common Algorithms – Element Separators

When you display the elements of a vector, you usually 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.

Common Algorithms – Element Separators

Print the separator before each element *except the initial one* (with index 0):

```
for (int i = 0; i < data.size(); i++)
{
   if (i > 0)
   {
      cout << " | ";
   }
   cout << data[i];
}
```

Common Algorithms – Counting Matches

How many elements of a vector fulfill a particular criterion?

Keep a counter and increment it for each matching element.

For example, this loop counts how many elements are greater than 100.

```
int count = 0;
for (int i = 0; i < data.size(); i++)
{
   if (data[i] > 100)
   {
      count++;
   }
}
```

Common Algorithms – Counting Matches

Which elements fulfill a criterion?

Use a second vector to collect the matches.

Here we collect all elements that are greater than 100.

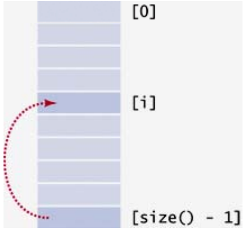
```
vector<double> matches;
for (int i = 0; i < data.size(); i++)
{
   if (data[i] > 100)
   {
      matches.push_back(data[i]);
   }
}
```

Common Algorithms – Removing an Element, Unordered

Suppose you want to remove the element at index **`i`**.

If the elements in the vector are not in any particular order, that task is easy to accomplish.

Simply overwrite the element to be removed with the *last* element of the vector, then shrink the size of the vector by removing the value that was copied.



```
int last_pos = data.size() - 1;
data[i] = data[last_pos];
data.pop_back();
```

Common Algorithms – Removing an Element, Ordered

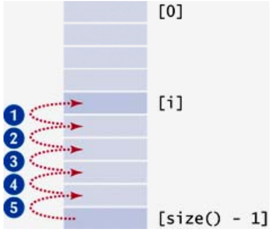
The situation is more complex if the order of the elements matters.

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

Common Algorithms – Removing an Element, Ordered

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



Common Algorithms – Inserting an Element Unordered

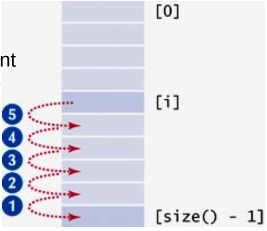
If the order of the elements does not matter, you can simply insert new elements at the end, using the **`push_back`** member function.

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 “up”.

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



Common Algorithms – Inserting an Element Ordered

To being this process, add a new element at the “top” of the vector by copying the last element to that new position.

Then do the shift.

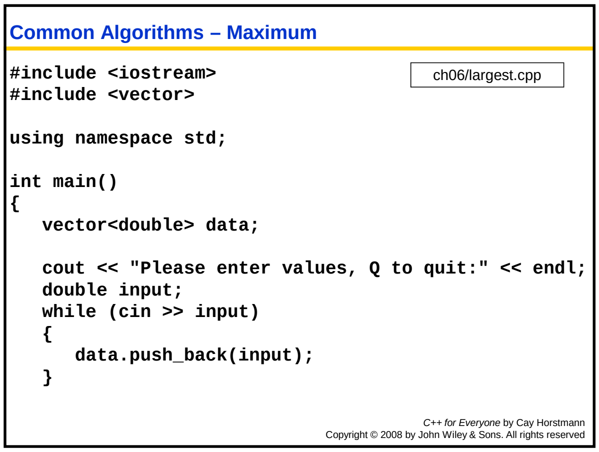
```
int last_pos = data.size() - 1;

// make room at the top
data.push_back(data[last_pos]);

// shift
for (int i = last_pos; i > pos; i--)
   data[i] = data[i - 1];

// insert
data[pos] = new_element;
```

Common Algorithms – Maximum

Who’s the tallest in the line?

Common Algorithms – Maximum

The following program will take user input (everyone’s height) and determine the largest value.

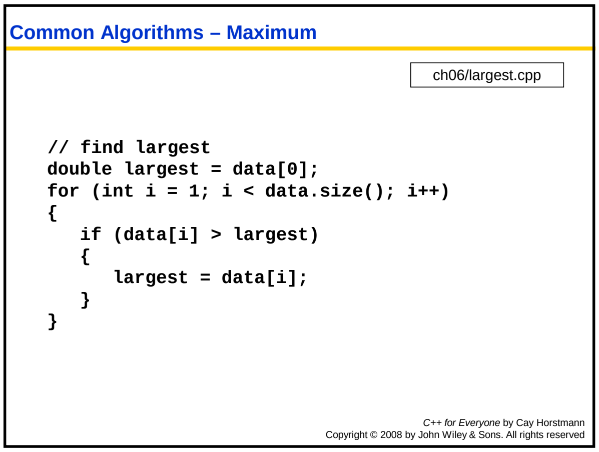


Common Algorithms – Maximum

ch06/largest.cpp

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector<double> data;

   cout << "Please enter values, Q to quit:" << endl;
   double input;
   while (cin >> input)
   {
      data.push_back(input);
   }
```

C++ for Everyone by Cay Horstmann

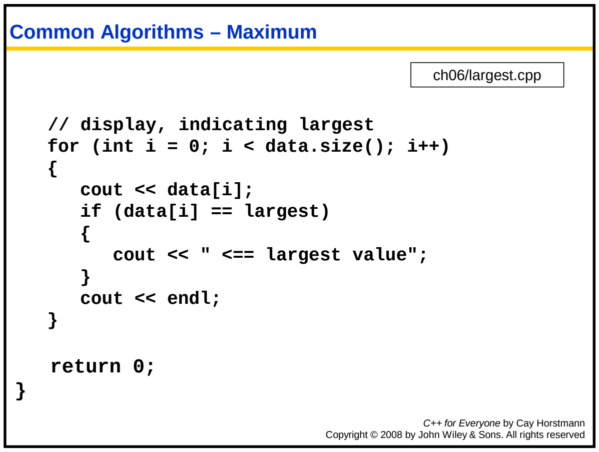


Common Algorithms – Maximum

ch06/largest.cpp

```
   // find largest
   double largest = data[0];
   for (int i = 1; i < data.size(); i++)
   {
      if (data[i] > largest)
      {
         largest = data[i];
      }
   }
```

C++ for Everyone by Cay Horstmann

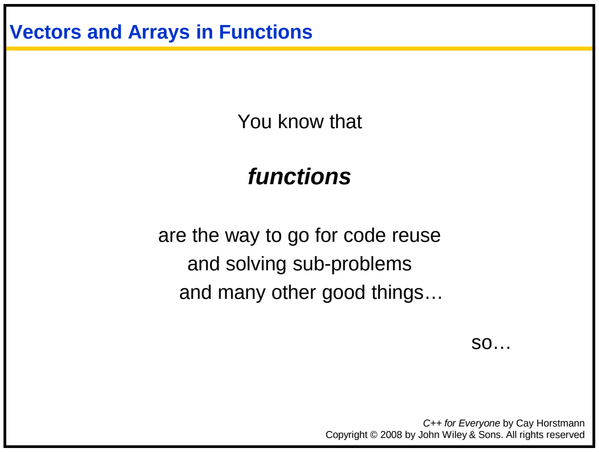


Common Algorithms – Maximum

ch06/largest.cpp

```
   // display, indicating largest
   for (int i = 0; i < data.size(); i++)
   {
      cout << data[i];
      if (data[i] == largest)
      {
         cout << " <== largest value";
      }
      cout << endl;
   }

   return 0;
}
```

C++ for Everyone by Cay Horstmann



Vectors and Arrays in Functions

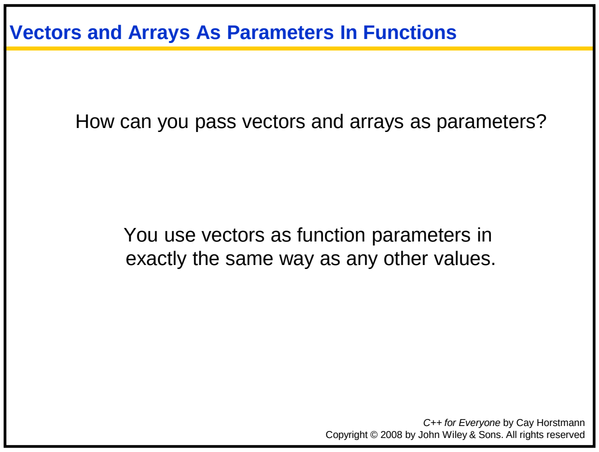
You know that

functions

are the way to go for code reuse
and solving sub-problems
and many other good things…

so…

C++ for Everyone by Cay Horstmann

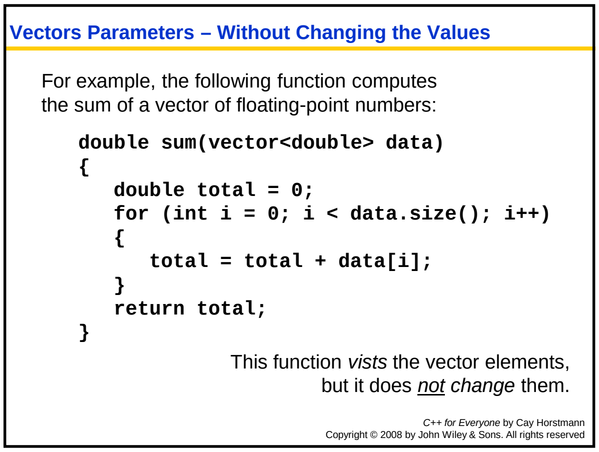


Vectors and Arrays As Parameters In Functions

How can you pass vectors and arrays as parameters?

You use vectors as function parameters in exactly the same way as any other values.

C++ for Everyone by Cay Horstmann



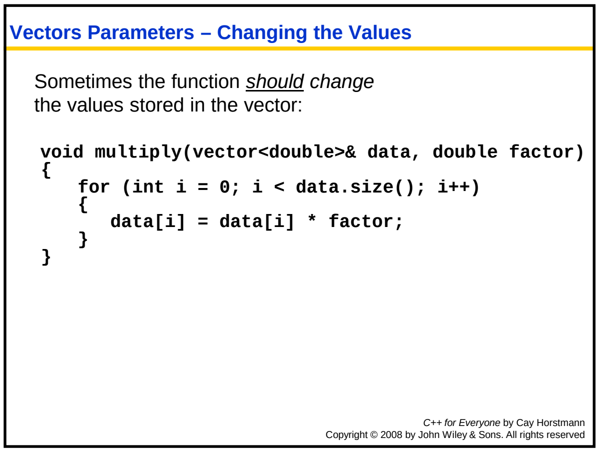
Vectors Parameters – Without Changing the Values

For example, the following function computes the sum of a vector of floating-point numbers:

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

This function *vists* the vector elements,
but it does *not* *change* them.

C++ for Everyone by Cay Horstmann

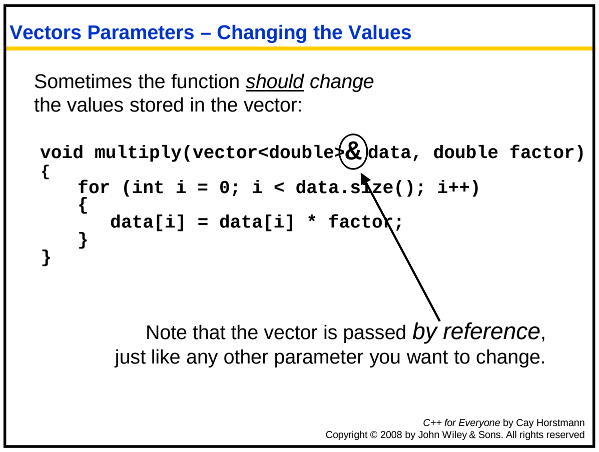


Vectors Parameters – Changing the Values

Sometimes the function *should* *change*
the values stored in the vector:

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

C++ for Everyone by Cay Horstmann



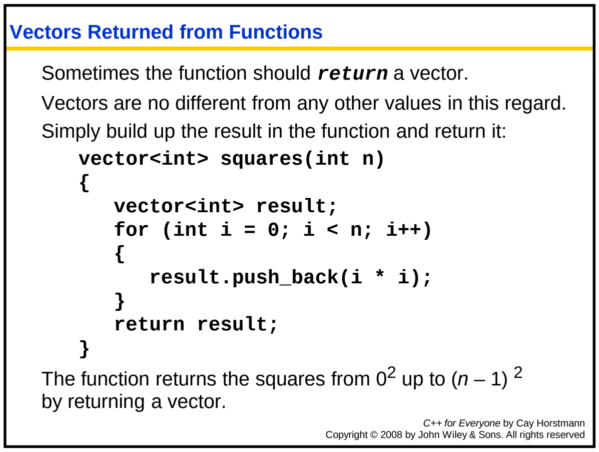
Vectors Parameters – Changing the Values

Sometimes the function *should* *change*
the values stored in the vector:

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

Note that the vector is passed *by reference*,
just like any other parameter you want to change.

C++ for Everyone by Cay Horstmann



Vectors Returned from Functions

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

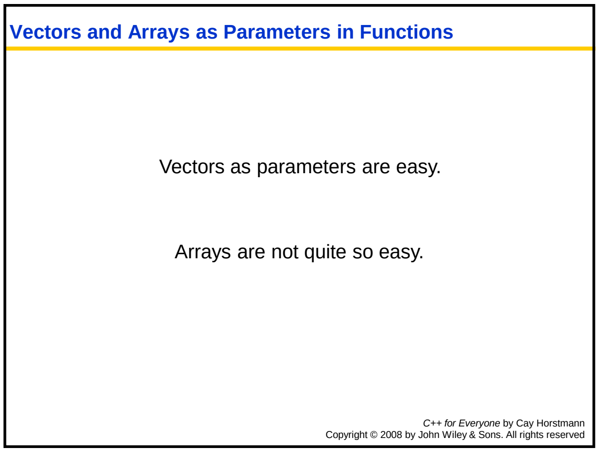
Vectors are no different from any other values in this regard.

Simply 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 $(n - 1)^2$
by returning a vector.

C++ for Everyone by Cay Horstmann

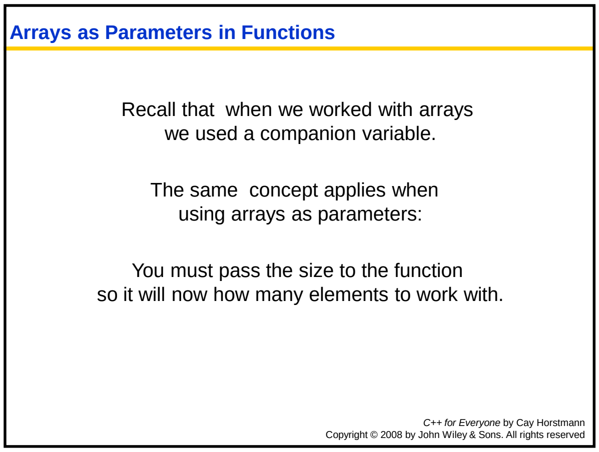


Vectors and Arrays as Parameters in Functions

Vectors as parameters are easy.

Arrays are not quite so easy.

C++ for Everyone by Cay Horstmann



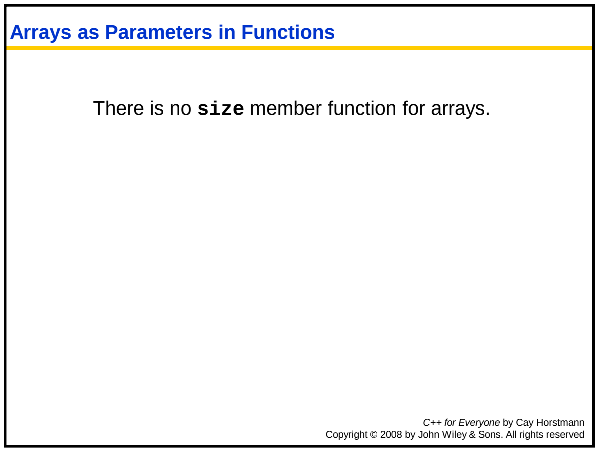
Arrays as Parameters in Functions

Recall that when we worked with arrays
we used a companion variable.

The same concept applies when
using arrays as parameters:

You must pass the size to the function
so it will now how many elements to work with.

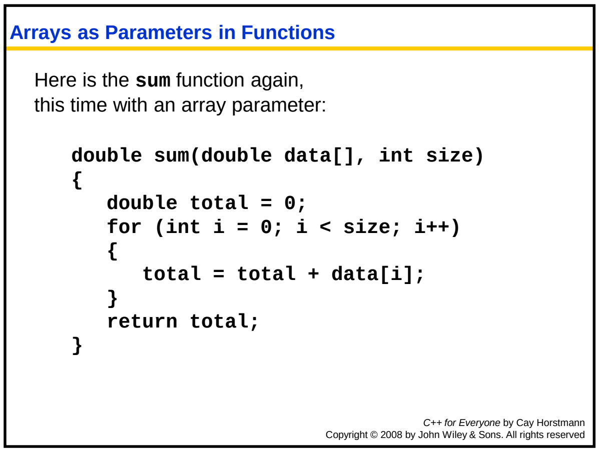
C++ for Everyone by Cay Horstmann



Arrays as Parameters in Functions

There is no **`size`** member function for arrays.

C++ for Everyone by Cay Horstmann

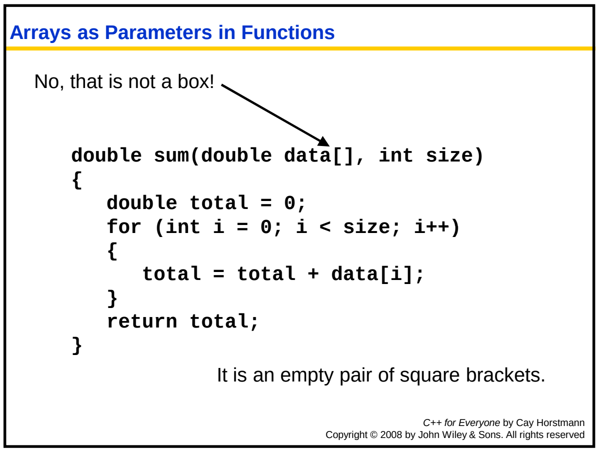


Arrays as Parameters in Functions

Here is the **sum** function again,
this time with an array parameter:

```
double sum(double data[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++)
   {
      total = total + data[i];
   }
   return total;
}
```

C++ for Everyone by Cay Horstmann



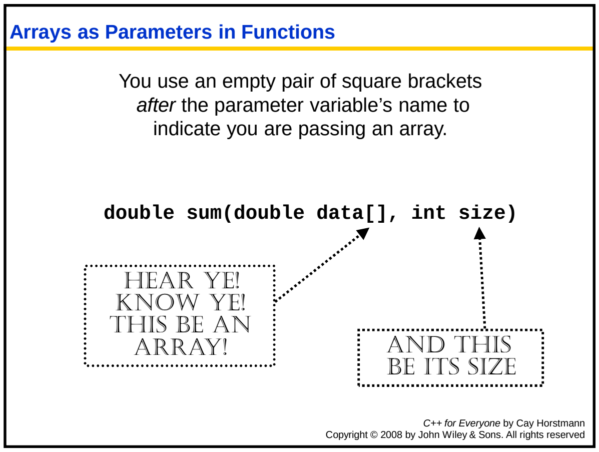
Arrays as Parameters in Functions

No, that is not a box!

```
double sum(double data[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++)
   {
      total = total + data[i];
   }
   return total;
}
```

It is an empty pair of square brackets.

C++ for Everyone by Cay Horstmann



Arrays as Parameters in Functions

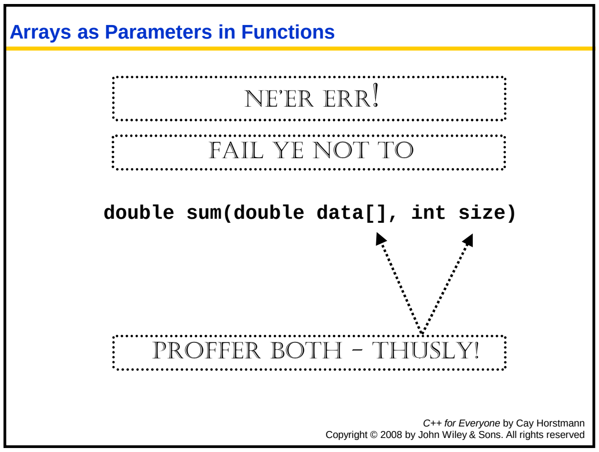
You use an empty pair of square brackets
after the parameter variable's name to
indicate you are passing an array.

```
double sum(double data[], int size)
```

HEAR YE! KNOW YE! THIS BE AN ARRAY!

AND THIS BE ITS SIZE

C++ for Everyone by Cay Horstmann



Arrays as Parameters in Functions

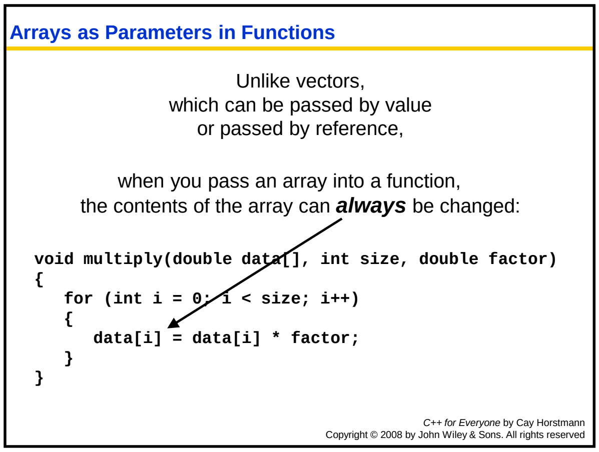
NE'ER ERR!

FAIL YE NOT TO

```
double sum(double data[], int size)
```

PROFFER BOTH – THUSLY!

C++ for Everyone by Cay Horstmann



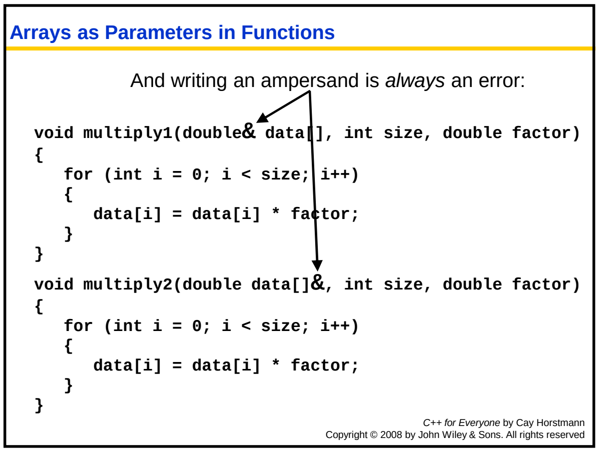
Arrays as Parameters in Functions

Unlike vectors,
which can be passed by value
or passed by reference,

when you pass an array into a function,
the contents of the array can ***always*** be changed:

```
void multiply(double data[], int size, double factor)
{
   for (int i = 0; i < size; i++)
   {
      data[i] = data[i] * factor;
   }
}
```

C++ for Everyone by Cay Horstmann



Arrays as Parameters in Functions

And writing an ampersand is *always* an error:

```
void multiply1(double& data[], int size, double factor)
{
   for (int i = 0; i < size; i++)
   {
      data[i] = data[i] * factor;
   }
}
void multiply2(double data[]&, int size, double factor)
{
   for (int i = 0; i < size; i++)
   {
      data[i] = data[i] * factor;
   }
}
```

C++ for Everyone by Cay Horstmann

Arrays as Parameters in Functions

And writing an ampersand is *always* an error:

```
void multiply1(double~~&~~ data[], int size, double factor)
{
   for (int i = 0; i < size; i++)
   {
      data[i] = data[i] * factor;
   }
}
void multiply2(double data[]~~&~~, int size, double factor)
{
   for (int i = 0; i < size; i++)
   {
      data[i] = data[i] * factor;
   }
}
```

Arrays as Parameters in Functions

And also unlike vectors,

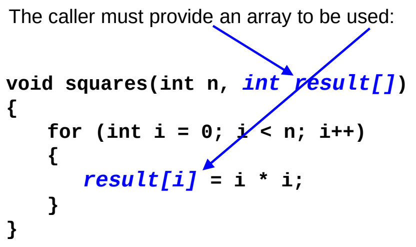
you cannot return an array

Arrays as Parameters in Functions

You cannot return an array.

```
??? squares(int n)
{
   int result[]
   for (int i = 0; i < n; i++)
   {
      result[i] = i * i;
   }
   return result; // ERROR
}
```

Arrays as Parameters in Functions



The caller must provide an array to be used:

```
void squares(int n, int result[])
{
   for (int i = 0; i < n; i++)
   {
      result[i] = i * i;
   }
}
```

A Program Using Vectors as Parameters

The following example program
reads values from standard input,
doubles them,
and prints the result.

The program uses three functions:

- **read_inputs** function returns a vector
- **multiply** has a vector as a reference parameter
- **print** has a vector as a value parameter

ch06/largest.cpp

```
#include <iostream>
#include <vector>

using namespace std;
```

A Program Using Vectors as Parameters

ch06/largest.cpp

```
/**
   Reads a sequence of floating-point numbers.
   @return a vector containing the numbers
*/
vector<double> read_inputs()
{
   vector<double> result;
   cout << "Please enter values, Q to quit:" << endl;
   bool more = true;
   while (more)
   {
      double input;
      cin >> input;
      if (cin.fail())
      {
         more = false;
      }
      else
      {
         result.push_back(input);
      }
   }
   return result;
}
```

A Program Using Vectors as Parameters

ch06/largest.cpp

```
/**
   Multiplies all elements of a vector by a factor
   @param data = a vector
   @param factor = the value with which element is multiplied
 */
void multiply(vector<double>& data, double factor)
{
   for (int i = 0; i < data.size(); i++)
   {
      data[i] = data[i] * factor;
   }
}
```

A Program Using Vectors as Parameters

ch06/largest.cpp

```
/**
   Prints the elements of a vector, separated by commas.
   @param data = a vector
 */
void print(vector<double> data)
{
   for (int i = 0; i < data.size(); i++)
   {
      if (i > 0) cout << ", ";
      cout << data[i];
   }
   cout << endl;
}
```

A Program Using Vectors as Parameters

ch06/largest.cpp

```
int main()
{
   vector<double> values = read_inputs();
   multiply(values, 2);
   print(values);

   return 0;
}
```

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.

Two-Dimensional Arrays

An arrangement consisting of *tabular data*:
rows and columns of values

is called:
a ***two-dimensional array***, or a ***matrix***.

Two-Dimensional Arrays

Consider this data from the 2006 Olympic skating competitions:

	Gold	Silver	Bronze
Canada	0	0	1
China	0	1	1
Japan	1	0	0
Russia	3	0	1
Switzerland	0	1	0
Ukraine	0	0	1
United States	0	2	0

Defining Two-Dimensional Arrays

C++ uses an array with *two* subscripts
to store a *two*-dimensional array.

```
const int COUNTRIES = 7;
const int MEDALS = 3;
int counts[COUNTRIES][MEDALS];
```

An array with 7 rows and 3 columns.
is suitable for storing our medal count data:

Defining Two-Dimensional Arrays – Unchangeable Size

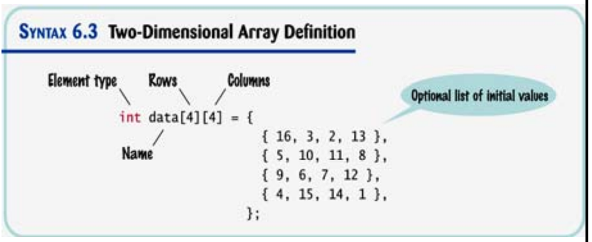
Just as with one-dimensional arrays,
you *cannot* change the size of
a two-dimensional array once it has been defined.

Defining Two-Dimensional Arrays – Initializing

But you can initialize a 2-D array:

```
int counts[COUNTRIES][MEDALS] =
   {
      { 0, 0, 1 },
      { 0, 1, 1 },
      { 1, 0, 0 },
      { 3, 0, 1 },
      { 0, 1, 0 },
      { 0, 0, 1 },
      { 0, 2, 0 }
   };
```

Defining Two-Dimensional Arrays – Syntax

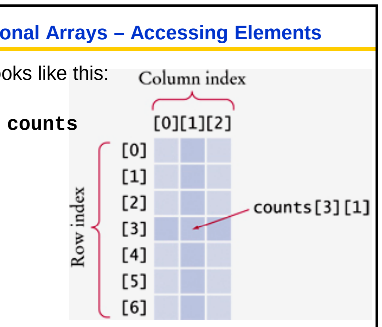


Syntax 6.3 Two-Dimensional Array Definition

```
int data[4][4] = {
                    { 16, 3, 2, 13 },
                    { 5, 10, 11, 8 },
                    { 9, 6, 7, 12 },
                    { 4, 15, 14, 1 },
                 };
```

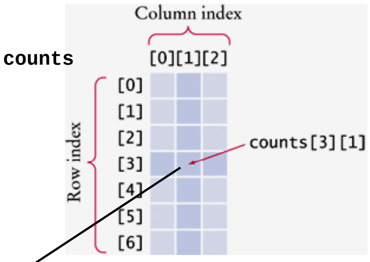
Defining Two-Dimensional Arrays – Accessing Elements

The Olympic array looks like this:



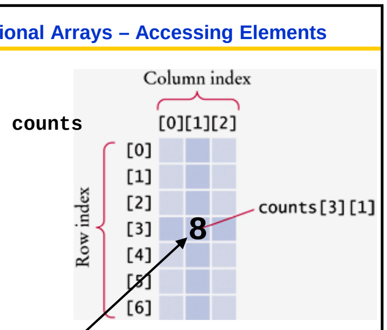
Access to the second element in the fourth row is:
`counts[3][1]`

Defining Two-Dimensional Arrays – Accessing Elements



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

Defining Two-Dimensional Arrays – Accessing Elements



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

Two-Dimensional Arrays

```
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;
}
```

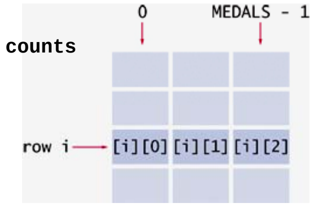
Computing Row and Column Totals

A common task is to compute row or column totals.

In our example,
the row totals give us the total number
of medals won by a particular country.

Computing Row and Column Totals

We must be careful to get the right indices.

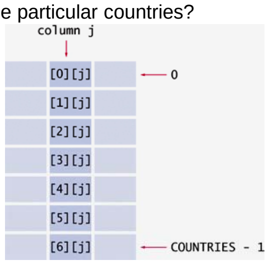


For each row **i**, we must use the column indices:
0, 1, … (MEDALS -1)

Computing Row and Column Totals

How many of each kind of medal was
won by the set of these particular countries?

counts



That would be a column total.

Let **j** be the silver column:

```
int total = 0;
for (int i = 0; i < COUNTRIES; i++)
{
   total = total + counts[i][j];
}
```

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.

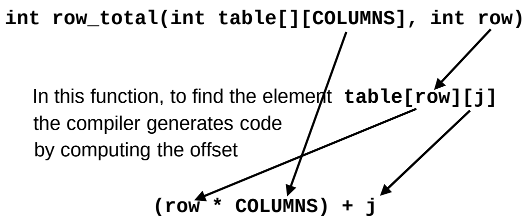
```
table[][COLUMNS]
```

Two-Dimensional Array Parameters

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;
}
```

Two-Dimensional Array Parameters



`int row_total(int table[][COLUMNS], int row)`

In this function, to find the element **`table[row][j]`**
the compiler generates code
by computing the offset

`(row * COLUMNS) + j`

Two-Dimensional Array Parameters

That function works for only arrays of 3 columns.

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

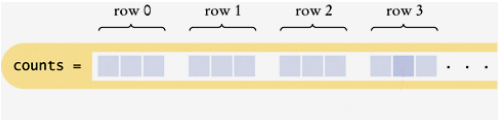
you would have to write
a different function
that has 4 as the parameter.

Hm.

Two-Dimensional Array Parameters

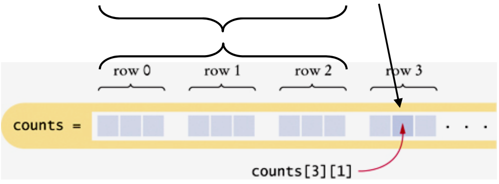
What's the reason behind this?

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



Two-Dimensional Array Parameters

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



Two-Dimensional Array Parameters

`int row_total(int table[][COLUMNS], int row)`

`table[]` looks like a normal 1D array.

Notice the empty square brackets.

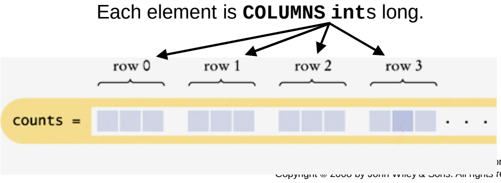
Two-Dimensional Array Parameters

```
int row_total(int table[][COLUMNS], int row)
```

`table[]` looks like a normal 1D array.

It is!

Each element is `COLUMNS` `int`s long.



Two-Dimensional Array Parameters

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;
}
```

Two-Dimensional Array Parameters – Common Error

Leaving out the columns value is a very common error.

```
int row_total(int table[][], int row)
...
```

The compiler doesn't know how "long" each row is!

Two-Dimensional Array Parameters – Not an Error

Putting a value for the rows is not an error.

```
int row_total(int table[17][COLUMNS], int row)
...
```

The compiler just ignores whatever you place there.

Two-Dimensional Array Parameters – Not an Error

Putting a value for the rows is not an error.

```
int row_total(int table[17][COLUMNS], int row)
...
```

The compiler just ignores whatever you place there.

```
int row_total(int table[][COLUMNS], int row)
...
```

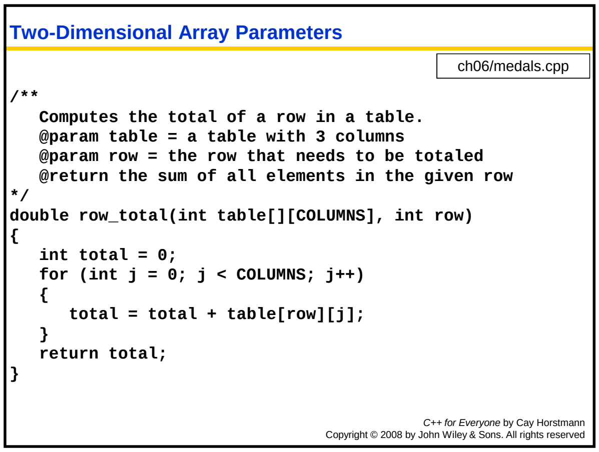
Never mind

Two-Dimensional Array Parameters

Here is the complete program for medal and column counts.

ch06/medals.cpp

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

const int COLUMNS = 3;
```

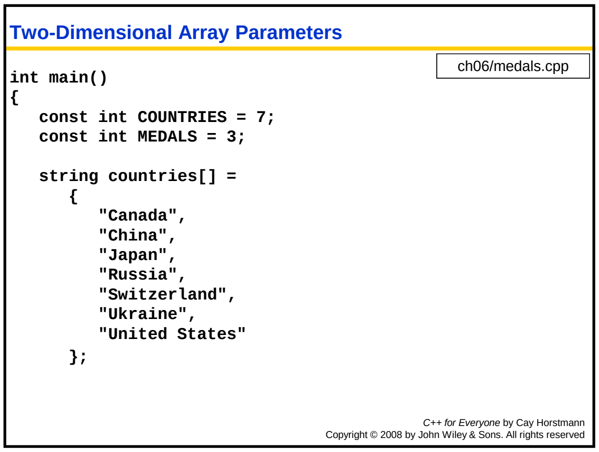


Two-Dimensional Array Parameters

ch06/medals.cpp

```
/**
   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;
}
```

C++ for Everyone by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved

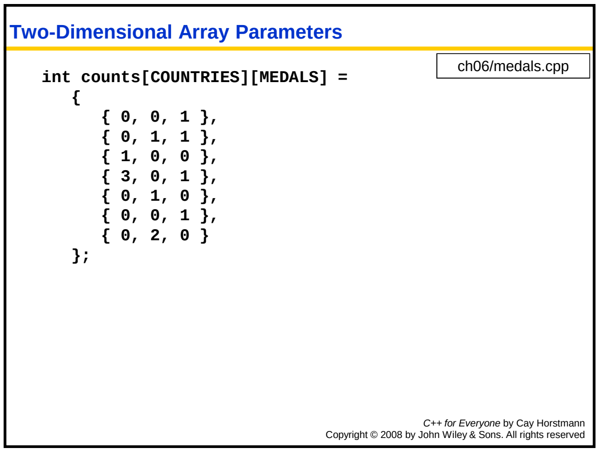


Two-Dimensional Array Parameters

ch06/medals.cpp

```
int main()
{
   const int COUNTRIES = 7;
   const int MEDALS = 3;

   string countries[] =
      {
         "Canada",
         "China",
         "Japan",
         "Russia",
         "Switzerland",
         "Ukraine",
         "United States"
      };
```

C++ for Everyone by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved

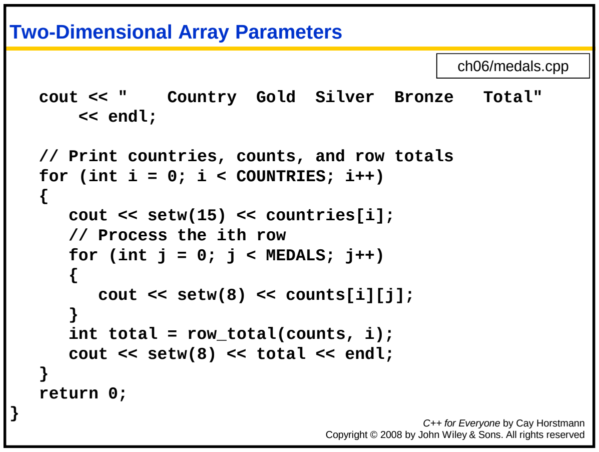


Two-Dimensional Array Parameters

ch06/medals.cpp

```
   int counts[COUNTRIES][MEDALS] =
      {
         { 0, 0, 1 },
         { 0, 1, 1 },
         { 1, 0, 0 },
         { 3, 0, 1 },
         { 0, 1, 0 },
         { 0, 0, 1 },
         { 0, 2, 0 }
      };
```

C++ for Everyone by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved

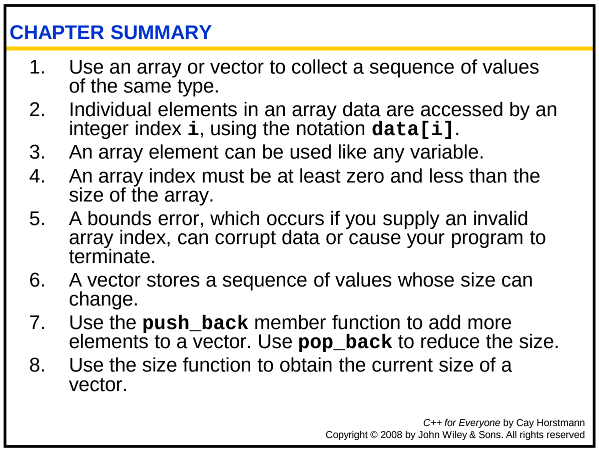


Two-Dimensional Array Parameters

ch06/medals.cpp

```
   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;
}
```

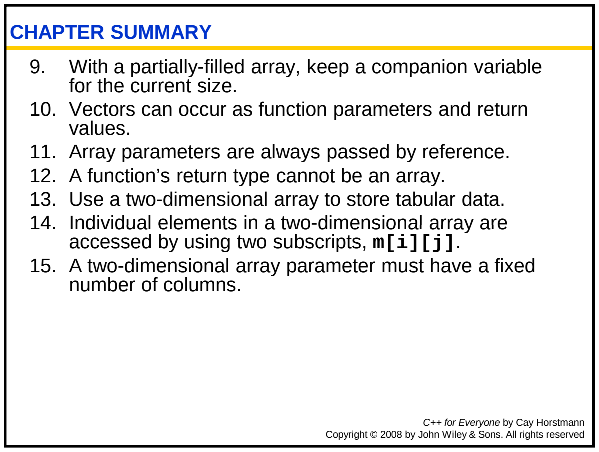
C++ for Everyone by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved



CHAPTER SUMMARY

1. Use an array or vector to collect a sequence of values of the same type.
2. Individual elements in an array data are accessed by an integer index **`i`**, using the notation **`data[i]`**.
3. An array element can be used like any variable.
4. An array index must be at least zero and less than the size of the array.
5. A bounds error, which occurs if you supply an invalid array index, can corrupt data or cause your program to terminate.
6. A vector stores a sequence of values whose size can change.
7. Use the **`push_back`** member function to add more elements to a vector. Use **`pop_back`** to reduce the size.
8. Use the size function to obtain the current size of a vector.

C++ for Everyone by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved



CHAPTER SUMMARY

9. With a partially-filled array, keep a companion variable for the current size.
10. Vectors can occur as function parameters and return values.
11. Array parameters are always passed by reference.
12. A function’s return type cannot be an array.
13. Use a two-dimensional array to store tabular data.
14. Individual elements in a two-dimensional array are accessed by using two subscripts, **`m[i][j]`**.
15. A two-dimensional array parameter must have a fixed number of columns.

C++ for Everyone by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved