Chapter Six: Arrays and Vectors

Slides by Lynn Gallagher

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

Using Vectors

- When you need to work with a large number of values – all together, the vector construct is your best choice.
  - Suppose you have the exam scores for a class of students and you want to (1) add 10 points to each of them, (2) find the max score, and (3) find the min score, then using a vector to store all of the exam scores is a good idea.
- 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

arrays do not grow automatically, they need to be told to grow

Using Arrays

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

Using Arrays and Vectors

In both vectors and arrays,
the stored data is of
the *same* data 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, of course)
(storable as `double`s)

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

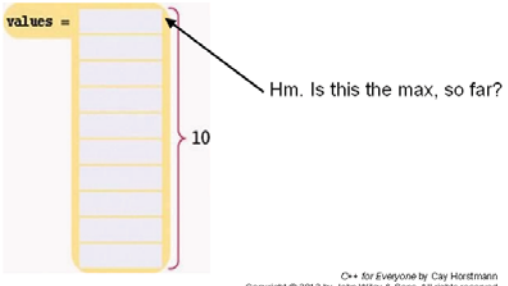
So would you create a variable for each?

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

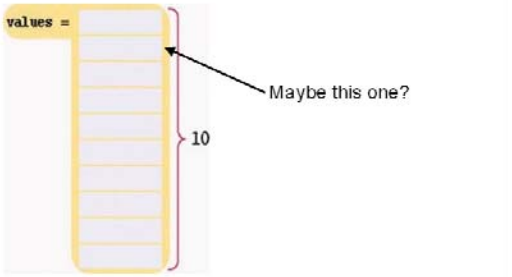
Then what ???

Using Arrays and Vectors

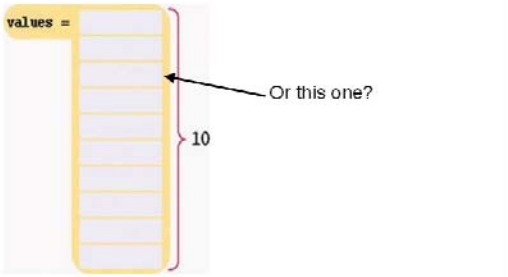
Instead, you could use construct where you can easily visit each element, checking and updating a variable holding the current maximum.



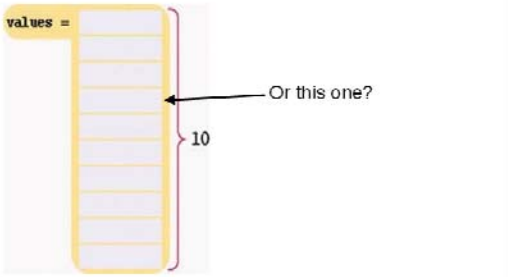
Using Arrays and Vectors



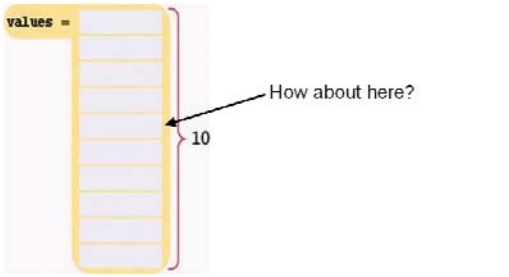
Using Arrays and Vectors



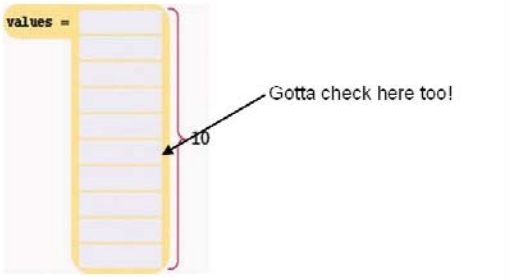
Using Arrays and Vectors



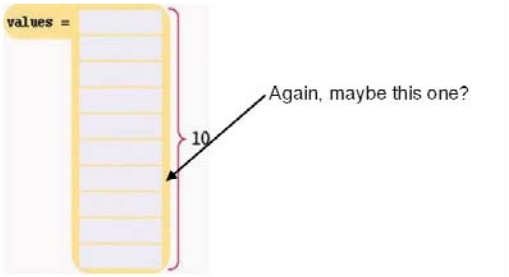
Using Arrays and Vectors



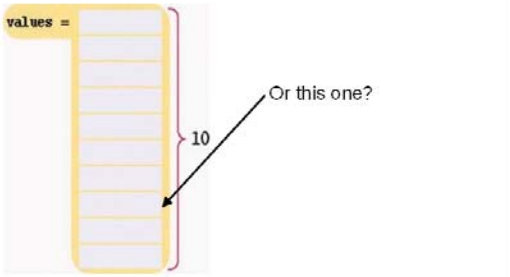
Using Arrays and Vectors



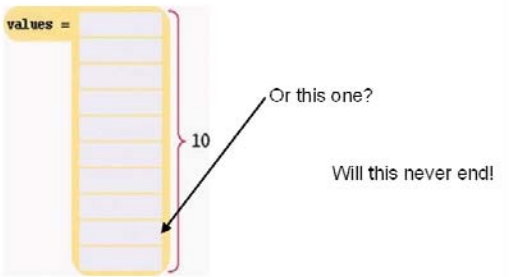
Using Arrays and Vectors



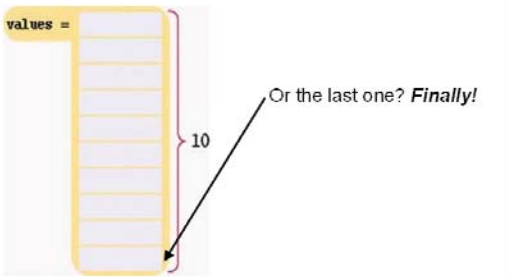
Using Arrays and Vectors



Using Arrays and Vectors



Using Arrays and Vectors



Using Arrays and Vectors

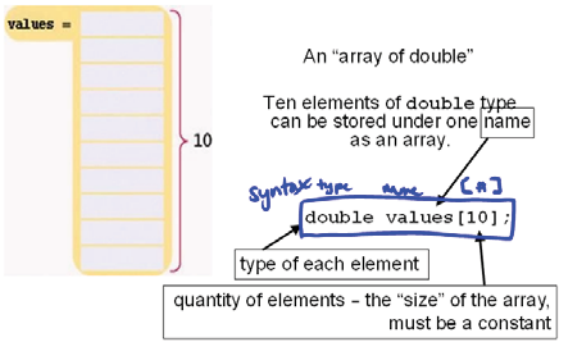
That would have been impossible with ten separate variables!

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

And what if there needed to be more data in the set?

ARGH!

Defining Arrays



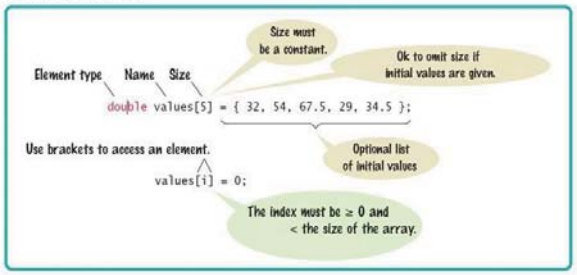
Defining Arrays with Initialization

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 };
```

Array Syntax

Defining an 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];`	**Caution:** In standard C++, the size must be a constant. This array definition 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.

Accessing an Array Element

An array element can be used like any variable.

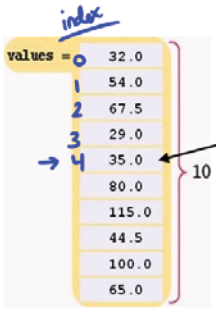
To access an array element, you use the notation:

`values[i]`

where `i` is the *index*.

Accessing an Array Element

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

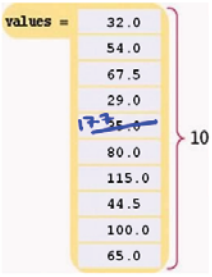



```
double values[10];
...
cout << values[4] << endl;
```

The output will be `35.0`.

Accessing an Array Element

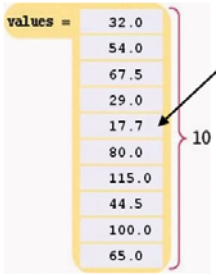
The same notation can be used to change the element.




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


Accessing an Array Element

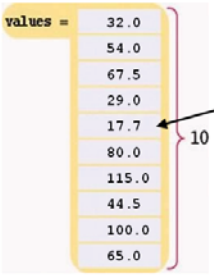
The same notation can be used to change the element.




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


Accessing an Array Element

The same notation can be used to change the element.

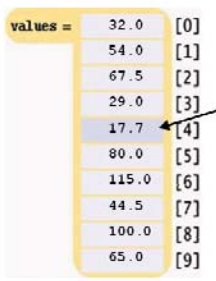



```
values[4] = 17.7;
cout << values[4] << endl;
```

The output will be `17.7`.

Accessing an Array Element

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




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

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

Accessing an Array Element

That is, the legal elements for the `values` array are:

`values[0]`, the ***first*** element
`values[1]`, the second element
`values[2]`, the third element
`values[3]`, the fourth element
`values[4]`, the fifth element
...
`values[9]`, the tenth ***and last legal*** element
recall: `double values[10];`

The index must be >= 0 and <= (size-1)

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

Partially-Filled Arrays

Suppose an array can hold 10 elements:

Does it always?
Just look at that beaker.
Guess not!

Partially-Filled Arrays – Capacity

How many elements, at most, can an array hold?

We call this quantity the *capacity*.

Partially-Filled Arrays – Capacity

For example, we may decide for a particular problem that there are usually ten or 11 values, but never more than 100.

We would set the capacity with a `const`:

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


Partially-Filled Arrays

Arrays will usually hold less than `CAPACITY` elements.

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?

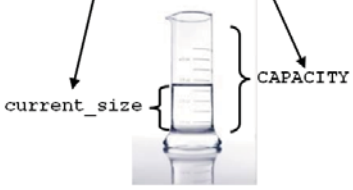
We will use a *companion variable* to hold that amount:

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

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

Suppose we add four elements to the array?

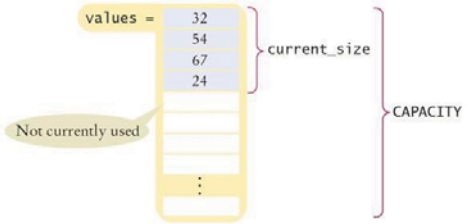
Partially-Filled Arrays – Companion Variable for Size

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

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



Partially-Filled Arrays – Companion Variable for Size

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

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



Partially-Filled Arrays – Capacity

The following loop fills an array with user input.
Each time the size of the array changes we update this variable:

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

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


Partially-Filled Arrays – Capacity

The following loop fills an array with user input.
Each time the size of the array changes we update this variable:

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

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


Partially-Filled Arrays – Capacity

When the loop ends, the companion variable `size` has the number of elements in the array.

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

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


Partially-Filled Arrays – Capacity

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

By using the `size` companion variable. not CAPACITY

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

When `i` is `0`, `values[i]` is `values[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 < CAPACITY; i++)
{
   cout << values[i] << endl;
}
```

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

When `i` is `1`,

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 < CAPACITY; i++)
{
   cout << values[1] << 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.

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 < CAPACITY; i++)
{
   cout << values[i] << 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`,

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 < CAPACITY; i++)
{
   cout << values[2] << 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.

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 < CAPACITY; i++)
{
   cout << values[i] << 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`,

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:

if CAPACITY is 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 ***last legal*** 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 < CAPACITY; i++)
{
   cout << values[i] << endl;
}
```

Note that the loop condition is that the index is

less than `CAPACITY`

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

But `CAPACITY` (10) ***is*** the number of elements we want to visit.

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];
```

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

DANGER!!!
DANGER!!!
DANGER!!!

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];
```

Clearly the meaning of each element is a score.

(even if it is a bad score, it's still a score)

Use Arrays for Sequences of Related Values

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 `double`s do *not* have the same meaning!

Use Arrays for Sequences of Related Values

But worse:

```
personal_data[   ] = new_shoe_size;
```


Use Arrays for Sequences of Related Values

But worse:

```
personal_data[ ? ] = new_shoe_size;
```

Oh dear!

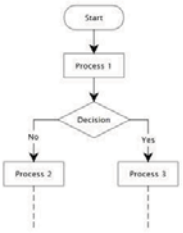
Which position was I using for the shoe size?

Use Arrays for Sequences of Related Values

Arrays should be used when the meaning of each element is the same.

Common Array and Vector Algorithms

There are many typical things that are done with sequences of values.



Next we share some common algorithms for processing values stored in both arrays and vectors.

(We will get to vectors a bit later but the algorithms are the same)

Common Algorithms – Filling an array with zeros

This loop fills an array with zeros:

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


Common Algorithms – Filling an array with squares

Here, we fill the array with squares (0, 1, 4, 9, 16, ...).

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

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

Common Algorithms – Copying one array into another

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**?

Common Algorithms – Copying the wrong way

Let's try what seems right and easy...

```
squares = lucky_numbers;
```

or `lucky_numbers = squares;`

...and wrong!

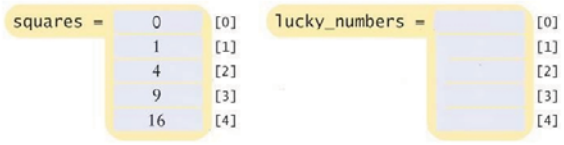
You cannot assign arrays!

You will have to do your own work, son.

Common Algorithms – Copying

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

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

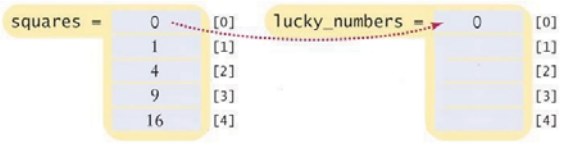
when **i** is **0**



Common Algorithms – Copying

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

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

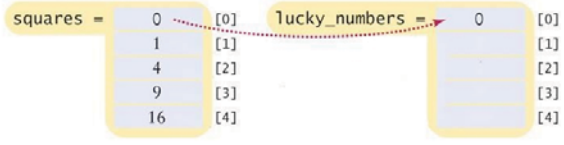
when **i** is **0**



Common Algorithms – Copying

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

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

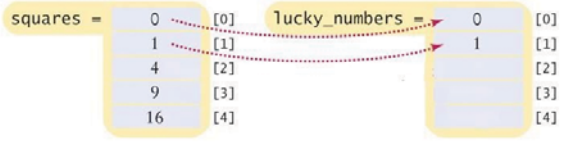
when **i** is **1**



Common Algorithms – Copying

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

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

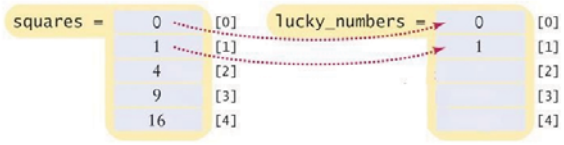
when **i** is **1**



Common Algorithms – Copying

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

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

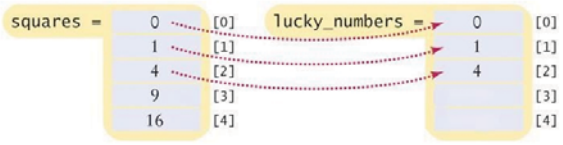
when `i` is 2



Common Algorithms – Copying

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

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

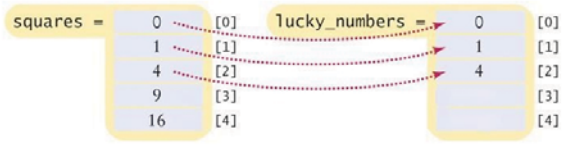
when `i` is 2



Common Algorithms – Copying

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

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

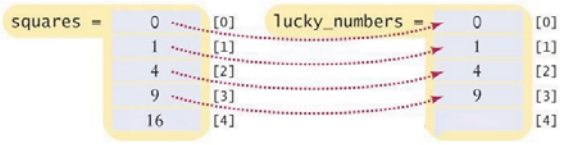
when `i` is 3



Common Algorithms – Copying

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

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

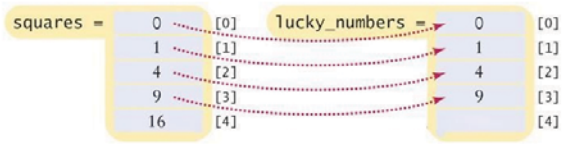
when `i` is 3



Common Algorithms – Copying

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

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

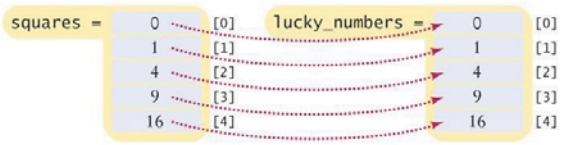
when `i` is 4



Common Algorithms – Copying

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

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

when `i` is 4



Method

suppose squares is already filled is 0, 1, 4, 9, 16

to copy the elements into lucky_numbers

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

Common Algorithms – Computing Sum and Average Value

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

total

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

← computes sum

The average is just arithmetic:

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

computes average ↗

Common Algorithms – Who Is the Tallest?

Who's the tallest in the line?

Common Algorithms – Who Is the Tallest?

If everyone's height is stored in an array, determining the largest value (who's the tallest person's height?) is just another algorithm...

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.

initialize largest to be first element

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


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

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

Common Algorithms – Who Is the Shortest?

Who's the shortest in the line?

Common Algorithms –Minimum

For the minimum, we just reverse the comparison.

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

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

Common Algorithms – Element Separators

When you display the elements of a vector, you 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.

To print five elements, you need *four* separators.

Common Algorithms – Element Separators

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

index
0
1 | 4 | 9 | 16 | 25

```
for (int i = 0; i < size of values; i++)
{
   if (i > 0)      or if (i>=1)
   {
      cout << " | ";
   }
   cout << values[i];
}
```

Common Algorithms – Linear Search

looking through an array for a specific value and want to know position that value is in

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

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

Don't get these tests in the wrong order!

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.