Chapter Six: Arrays and Vectors

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

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

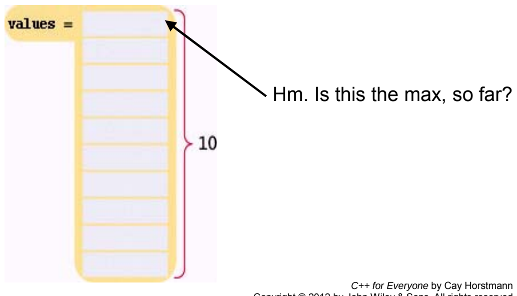
So would you create a variable for each?

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

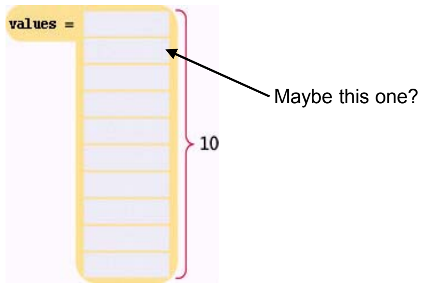
Then what ???

Using Arrays and Vectors

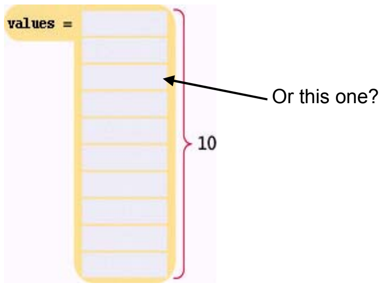
Instead, you could use construct where you can easily visit each element, checking and updating a variable holding the current maximum.



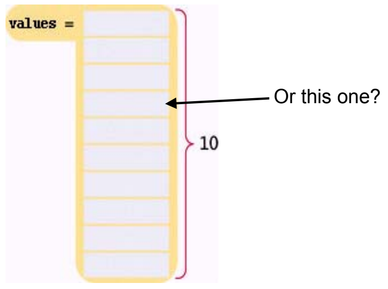
Using Arrays and Vectors



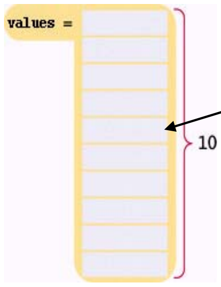
Using Arrays and Vectors



Using Arrays and Vectors

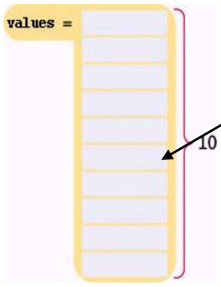


Using Arrays and Vectors



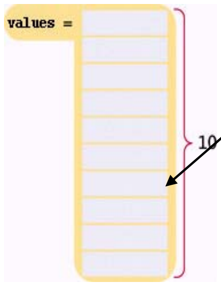
How about here?

Using Arrays and Vectors



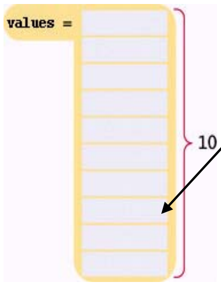
Gotta check here too!

Using Arrays and Vectors



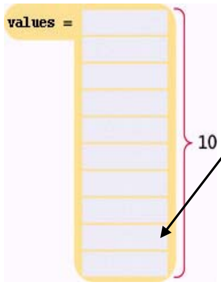
Again, maybe this one?

Using Arrays and Vectors



Or this one?

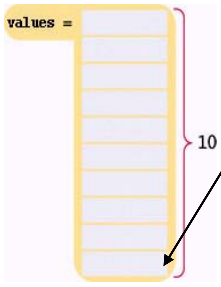
Using Arrays and Vectors



Or this one?

Will this never end!

Using Arrays and Vectors



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

Using Arrays and Vectors

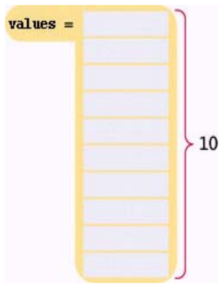
That would have been impossible with ten separate variables!

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

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

ARGH!

Defining Arrays



An "array of double"

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

```
double values[10];
```

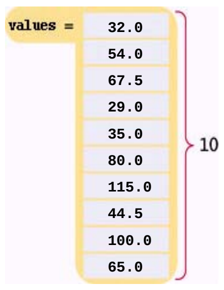
type of each element

quantity of elements – the "size" of the array, must be a constant

Defining Arrays with Initialization

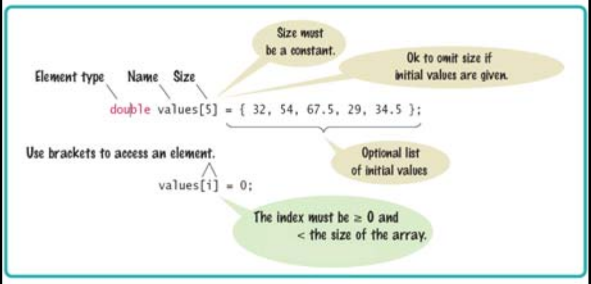
When you define an array, you can specify the initial values:

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



Array Syntax

Defining an Array


Size must be a constant.
Ok to omit size if initial values are given.
Element type
Name
Size
Use brackets to access an element.
Optional list of initial values
The index must be ≥ 0 and < the size of the array.

```
double values[5] = { 32, 54, 67.5, 29, 34.5 };
values[i] = 0;
```

Array Syntax

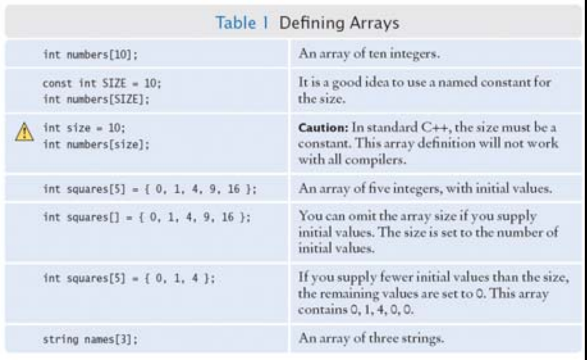


Table 1 Defining Arrays

`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.

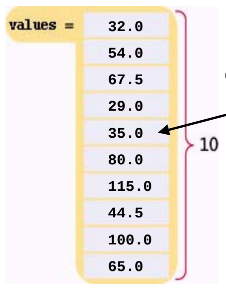
To access an array element, you use the notation:

`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*.

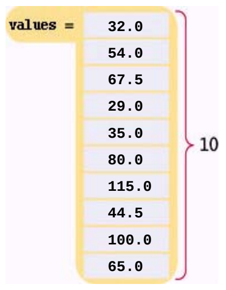


```
double values[10];
...
cout << values[4] << endl;
```

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

Accessing an Array Element

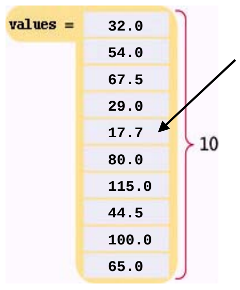
The same notation can be used to change the element.



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

Accessing an Array Element

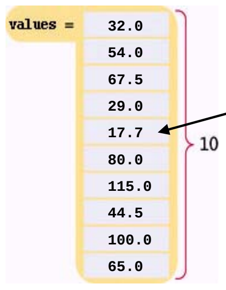
The same notation can be used to change the element.



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

Accessing an Array Element

The same notation can be used to change the element.

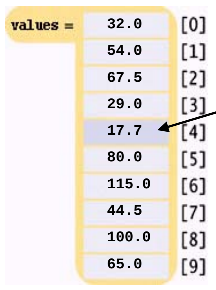


```
values[4] = 17.7;
cout << values[4] << endl;
```

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 **`<= 9`**.
0, 1, 2, 3, 4, 5, 6, 7, 8, 9 is 10 numbers.

Partially-Filled Arrays

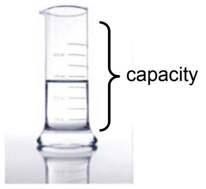
Suppose an array can hold 10 elements:

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.

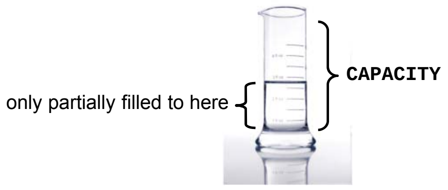
We would set the capacity with a **`const`**:

```
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?

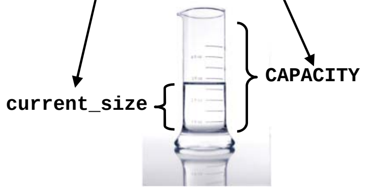
We will use a *companion variable* to hold that amount:

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

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

Suppose we add four elements to the array?

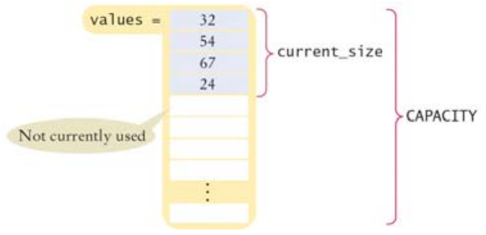
Partially-Filled Arrays – Companion Variable for Size

```
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] = x;
      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] = x;
      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] = x;
      size++;
   }
}
```

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 << 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:

```
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 data[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!

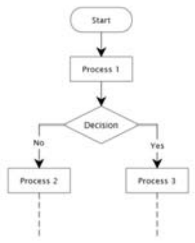
Which position was I using for the shoe size?

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;
```

…and wrong!

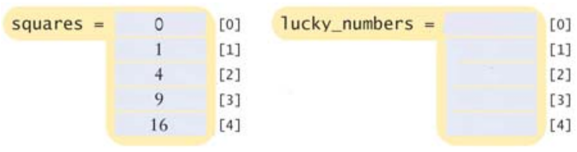
You cannot assign arrays!

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;
}
```

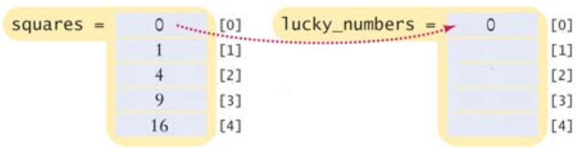
when **`i`** is **`0`**



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;
}
```

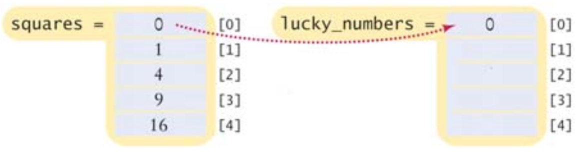
when **`i`** is **`0`**



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;
}
```

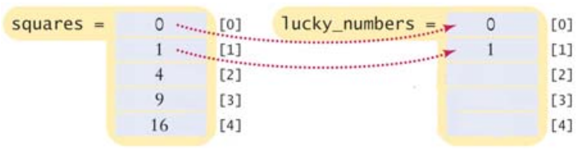
when **`i`** is **`1`**



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;
}
```

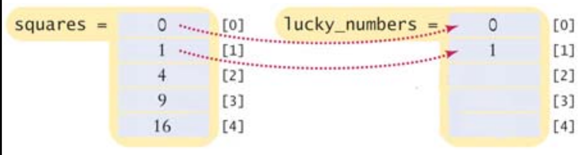
when **`i`** is **`1`**



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;
}
```

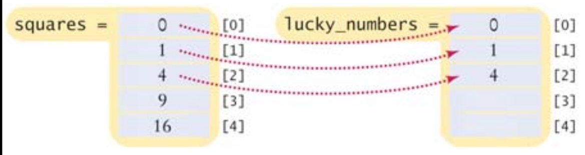
when **i** is **2**



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;
}
```

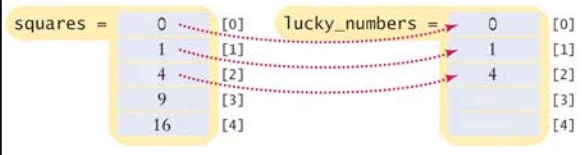
when **i** is **2**



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;
}
```

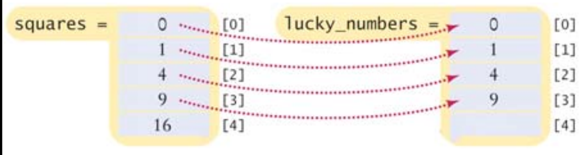
when **i** is **3**



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;
}
```

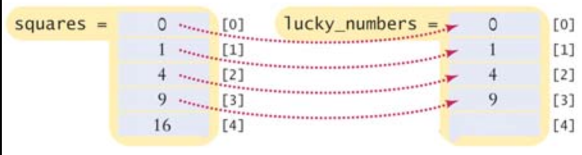
when **i** is **3**



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;
}
```

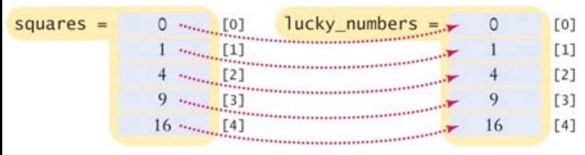
when **i** is **4**



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**



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.

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

The average is just arithmetic:

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

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.

```
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):

```
1 | 4 | 9 | 16 | 25
```

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

Common Algorithms – Linear Search

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

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

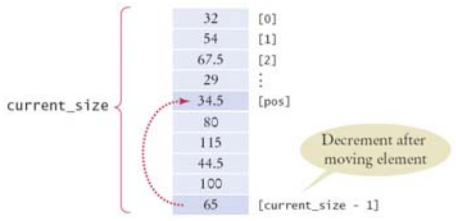
Don't get these tests in the wrong order!

Common Algorithms – Removing an Element, Unordered

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

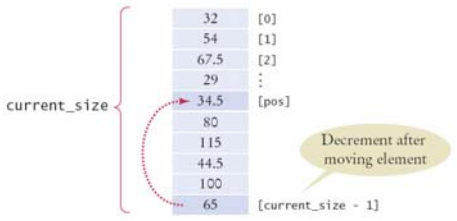
If the elements in the vector are not in any particular order, that task is easy to accomplish.

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.



Common Algorithms – Removing an Element, Unordered

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



Common Algorithms – Removing an Element, Ordered

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

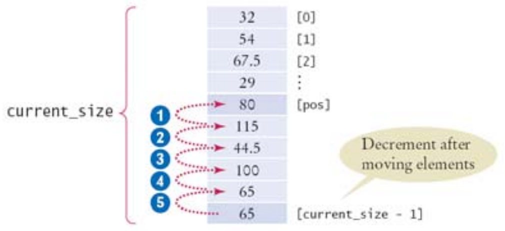
Then you must move all elements following the element to be removed “down” (to a lower index), and then shrink the size of the vector by removing the last element.

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

C++ for Everyone by Cay Horstmann

Common Algorithms – Removing an Element, Ordered

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

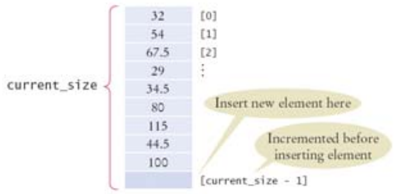


C++ for Everyone by Cay Horstmann

Common Algorithms – Inserting an Element Unordered

If the order of the elements does not matter, in a partially filled array (which is the only kind you can insert into), you can simply insert a new element at the end.

```
if (current_size < CAPACITY)
{
   current_size++;
   values[current_size - 1] = new_element;
}
```



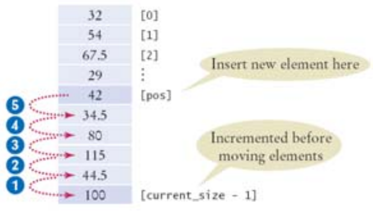
C++ for Everyone by Cay Horstmann

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]`**.



C++ for Everyone by Cay Horstmann

Common Algorithms – Inserting an Element Ordered

First, you must make the array one larger by incrementing **`current_size`**.

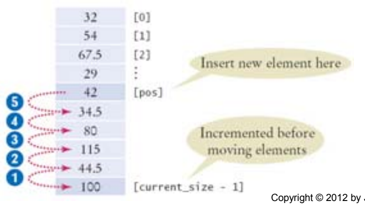
Next, move all elements above the insertion location to a higher index.

Finally, insert the new element in the place you made for it.

C++ for Everyone by Cay Horstmann

Common Algorithms – Inserting an Element Ordered

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

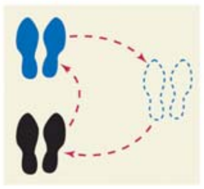


C++ for Everyone by Cay Horstmann

Common Algorithms – Swapping Elements

Swapping two elements in an array
is an important part of sorting an array.

To do a swap of two things,
you need *three* things!



Common Algorithms – Swapping Elements

Suppose we need to swap the values at
positions **`i`** and **`j`** in the array.
Will this work?

```
values[i] = values[j];
values[j] = values[i];
```

Look closely!

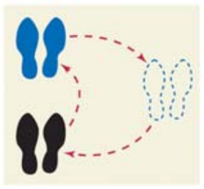
In the first line you lost – forever! – the value at **`i`**,
replacing it with the value at **`j`**.

Then what?
Put' **`j`**'s value back in **`j`** in the second line?

ARGHHH!

Common Algorithms – Swapping Elements

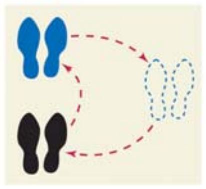
You need a *third* dance partner!



Common Algorithms – Swapping Elements

Let's
Waltz!

1, 2, 3,
go, 2, 3,



Common Algorithms – Swapping Elements

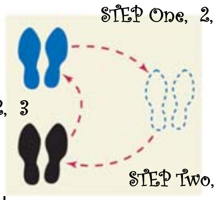
```
double temp = values[i];
values[i] = values[j];
values[j] = temp;
```

STEP One, 2, 3

save the
value at **`i`**

STEP Three, 2, 3

now you can
change the
value at **`j`**
because you
saved from **`i`**.



replace the
value at **`i`**

STEP Two, 2, 3

Common Algorithms – Reading Input

If the know how many input values the user will supply,
you can store them directly into the array:

```
double values[NUMBER_OF_INPUTS];
for (i = 0; i < NUMBER_OF_INPUTS; i++)
{
   cin >> values[i];
}
```

Common Algorithms – Reading Input

When there will be an arbitrary number of inputs,
things get more complicated.
But not hopeless.
Add values to the end of the array until all inputs have been made.
Again, the companion variable will have the number of inputs.

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

Common Algorithms – Reading Input

Unfortunately it's even more complicated:

Once the array is full, we allow the user to keep entering!

Because we can't change the size
of an array after it has been created,
we'll just have to give up for now.

Common Algorithms

Now back to where we started:

How do we determine the largest in a set of data?

HANDOUT – Example: largest.cpp

Sorting with the C++ Library

Getting data into order is something
that is often needed.

For Example:

- An alphabetical listing.
- A list of grades in descending order.

Sorting with the C++ Library

In C++, you call the **`sort`** function
to do your sorting for you.
But the syntax is new to you:

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

```
sort(values, values + current_size);
```

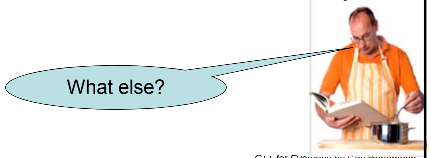
To sort the elements into ascending numerical order,
you call the **`sort`** algorithm:

Sorting with the C++ Library

Yes, we said *call* the **`sort`** *algorithm*.

C++ has a library named **`algorithm`** that contains…
algorithms,
as functions.

```
sort(values, values + current_size);
```



Sorting with the C++ Library

You will need to write:

```
#include <algorithm>
```

in order to use the **`sort`** function.

```
sort(values, values + current_size);
```

Sorting with the C++ Library

Notice also that you must tell the **`sort`** function
where to begin: **`values`**,
(which is the start of the array)
and where to end: **`values + current_size`**,
(which is one *after* the last element in the array).

```
sort(values, values + current_size);
```

Arrays as Parameters in Functions

Recall that when we work with arrays
we use a companion variable.

The same concept applies when
using arrays as parameters:

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

Arrays as Parameters in Functions

Here is the **`sum`** function with an array parameter:
Notice that to pass one array, it takes two parameters.

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

Arrays as Parameters in Functions

No, that is not a box!

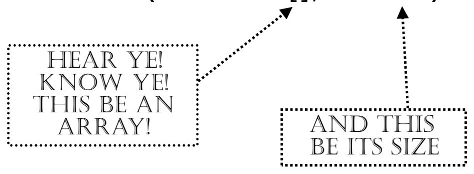
```
double sum(double data[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++)
   {
      total = total + data[i];
   }
   return total;
}
```

It is an empty pair of square brackets.

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)
```



Arrays as Parameters in Functions

NE'ER ERR!

FAIL YE NOT TO

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

PROFFER BOTH – THUSLY!

Arrays as Parameters in Functions

When you call the function,
supply both the name of the array and the size:

```
double NUMBER_OF_SCORES = 10;
double scores[NUMBER_OF_SCORES]
   = { 32, 54, 67.5, 29, 34.5, 80, 115, 44.5, 100, 65 };
double total_score = sum(scores, NUMBER_OF_SCORES);
```

You can also pass a smaller size to the function:

```
double partial_score = sum(scores, 5);
```

This will sum over only the first five **`double`**s in the array.

Arrays as Parameters in Functions

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

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

Arrays as Parameters in Functions

And writing an ampersand is *always* an error:

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

Arrays as Parameters in Functions

And writing an ampersand is *always* an error:

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

Arrays as Parameters in Functions

You can pass an array into a function

but

you cannot return an array.

Arrays as Parameters in Functions

If you cannot return an array, how can the caller get the data?

```
??? 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;
   }
}
```

Arrays as Parameters in Functions

A function can change the size of an array.
It should let the caller know of any change
by returning the new size.

```
int read_inputs(double inputs[], int capacity)
{
   int current_size = 0;
   double input;
   while (cin >> input)
   {
      if (current_size < capacity)
      {
         inputs[current_size] = input;
         current_size++;
      }
   }
   return current_size;
}
```

Arrays as Parameters in Functions

Here is a call to the function:

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

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

Arrays as Parameters in Functions

Or it can let the caller know by using a
reference parameter:

```
void append_inputs(double inputs[], int capacity,
                   int& current_size)
{
   double input;
   while (cin >> input)
   {
      if (current_size < capacity)
      {
         inputs[current_size] = input;
         current_size++;
      }
   }
}
```

Arrays as Parameters in Functions

Here is a call to the reference parameter
version of **`append_inputs`**:

```
append_inputs(values, MAXIMUM_NUMBER_OF_VALUES,
              current_size);
```

As before, after the call,
the **`current_size`** variable
specifies how many are in the array.

Arrays as Parameters in Functions

Our next program uses the preceding functions to read values from standard input, double them, and print the result.

- The **`read_inputs`** function fills an array with the input values. It returns the number of elements that were read.
- The **`multiply`** function modifies the contents of the array that it receives, demonstrating that arrays can be changed inside the function to which they are passed.
- The **`print`** function does not modify the contents of the array that it receives.

Problem Solving: Adapting Algorithms

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

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

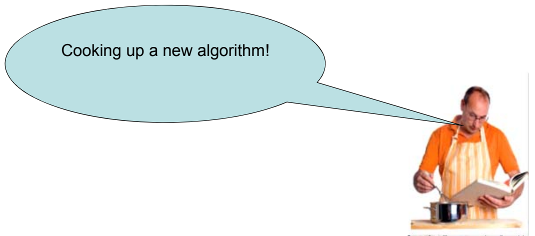
What to do?

No, "give up" is not an option!

Problem Solving: Adapting Algorithms

You can try to use algorithms you already know to produce a new algorithm that will solve this problem.

(Then you'll have yet another algorithm – even more!)



Problem Solving: Adapting Algorithms

Consider this problem:

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

but be nice:
drop the lowest score.

Problem Solving: Adapting Algorithms

Hmm, what do I know how to do?

Problem Solving: Adapting Algorithms

Calculate the sum:

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

Problem Solving: Adapting Algorithms

Find the minimum:

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

Problem Solving: Adapting Algorithms

Remove an element:

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

Problem Solving: Adapting Algorithms

Aha! Here is the algorithm:

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

Problem Solving: Adapting Algorithms

WAIT!

(Houston, we have a problem…)

Problem Solving: Adapting Algorithms

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

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

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

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

Problem Solving: Adapting Algorithms

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

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

Problem Solving: Adapting Algorithms

Aha! Here is the algorithm:

1. *Find the minimum*
2. *Find the position of the minimum*
   → the one I just searched for!!!
3. *Remove it from the array*
4. *Calculate the sum*
   (will be without the lowest score)
5. *Calculate the final score*

Problem Solving: Adapting Algorithms

But notice what I did…

I searched
for the minimum
and then
I searched
for the position…
…of the minimum!

Problem Solving: Adapting Algorithms

I wonder if I can *adapt* the algorithm
that finds the minimum so that it finds
the position of the minimum?

Problem Solving: Adapting Algorithms

I'll start with this:

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

Problem Solving: Adapting Algorithms

What is it about the minimum value
and where the minimum value is?

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

Problem Solving: Adapting Algorithms

What is it about the minimum value
and where the minimum value is?

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

Problem Solving: Adapting Algorithms

There it is!

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

Problem Solving: Adapting Algorithms

Finally:

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

Discovering Algorithms by Manipulating Physical Objects

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

What to do?

No, again, “give up” is not an option!

Discovering Algorithms by Manipulating Physical Objects

There is a technique that you can use called:

MANIPULATING PHYSICAL OBJECTS

better know as:

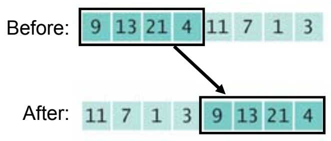
playing around with things.

Discovering Algorithms by Manipulating Physical Objects

Discovering Algorithms by Manipulating Physical Objects

Here is a problem:

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



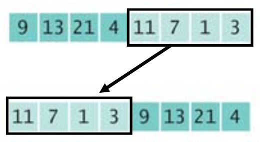
Discovering Algorithms by Manipulating Physical Objects

Here is a problem:

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

Before:

After:



Discovering Algorithms by Manipulating Physical Objects

Discovering Algorithms by Manipulating Physical Objects

To learn this *Manipulating Physical Objects* technique,
let’s play with some coins
and review some algorithms you already know.

OK, let’s *manipulate* some coins.
Go get eight coins.

Discovering Algorithms by Manipulating Physical Objects

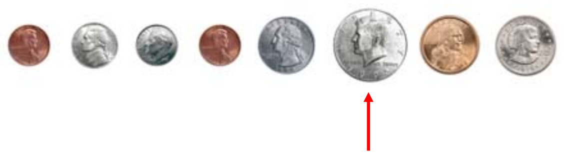
Good.

Discovering Algorithms by Manipulating Physical Objects

What algorithms do you know
that allow you to rearrange a set of coins?

Discovering Algorithms by Manipulating Physical Objects

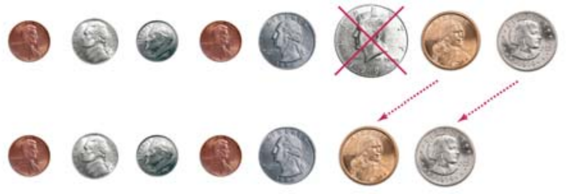
You know how to remove a coin.



You! Be gone!

Discovering Algorithms by Manipulating Physical Objects

You know how to remove a coin.



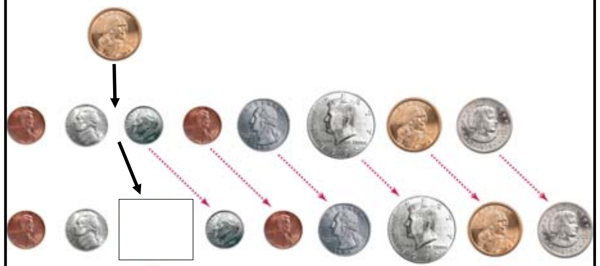
Discovering Algorithms by Manipulating Physical Objects

You know how to insert a coin at a specific position.

You!
Right there!

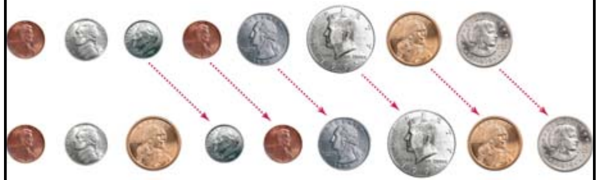
Discovering Algorithms by Manipulating Physical Objects

You know how to insert a coin at a specific position.



Discovering Algorithms by Manipulating Physical Objects

You know how to insert a coin at a specific position.



Discovering Algorithms by Manipulating Physical Objects

And you know how to swap two elements.

You two!
Swap places!

Discovering Algorithms by Manipulating Physical Objects

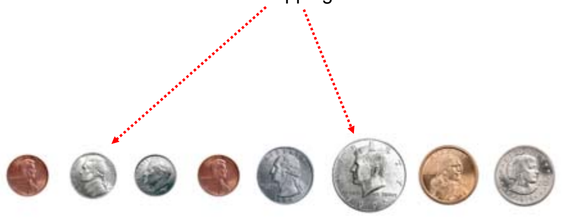
And you know how to swap two elements.

Discovering Algorithms by Manipulating Physical Objects

And you know how to swap two elements.

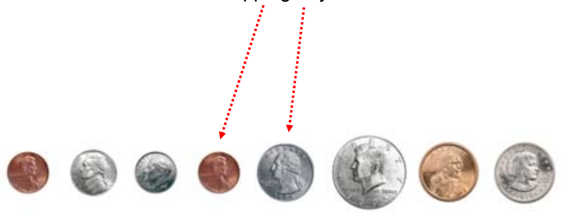
Discovering Algorithms by Manipulating Physical Objects

Swapping.



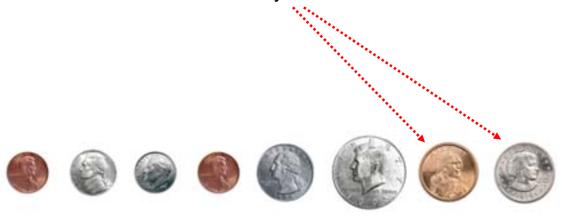
Discovering Algorithms by Manipulating Physical Objects

Swapping any two.



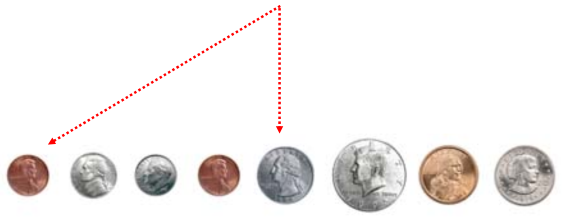
Discovering Algorithms by Manipulating Physical Objects

Any two.



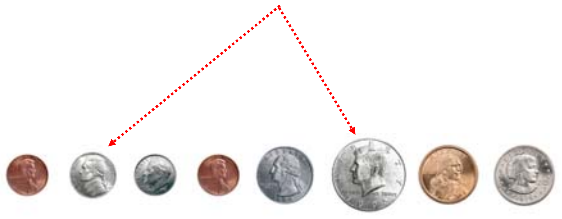
Discovering Algorithms by Manipulating Physical Objects

Hm.



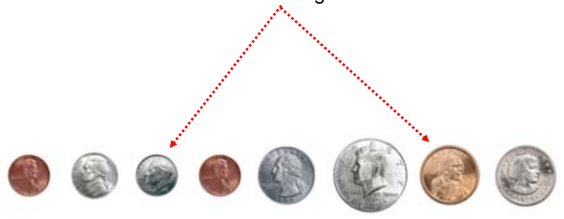
Discovering Algorithms by Manipulating Physical Objects

And hm.



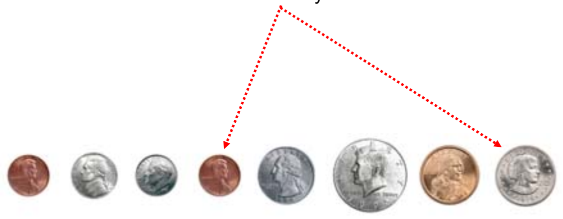
Discovering Algorithms by Manipulating Physical Objects

Then hm again.

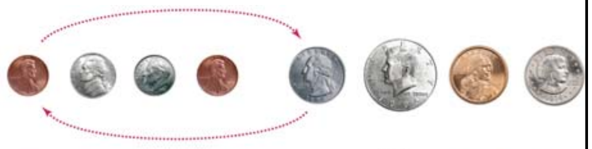


Discovering Algorithms by Manipulating Physical Objects

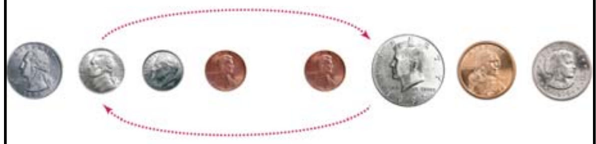
And finally…



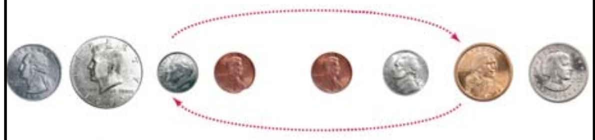
Discovering Algorithms by Manipulating Physical Objects



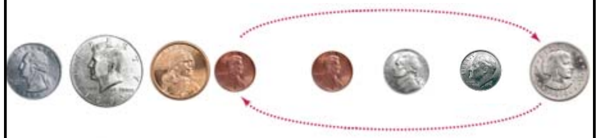
Discovering Algorithms by Manipulating Physical Objects



Discovering Algorithms by Manipulating Physical Objects

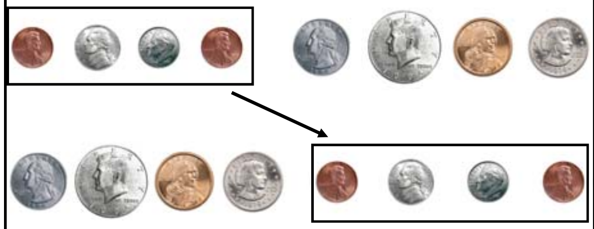


Discovering Algorithms by Manipulating Physical Objects

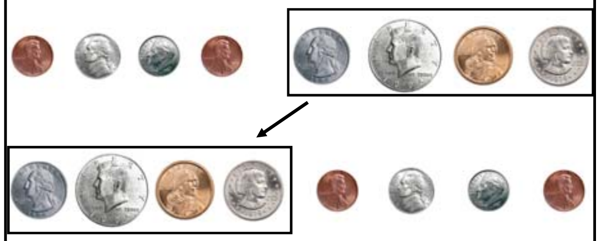


Discovering Algorithms by Manipulating Physical Objects

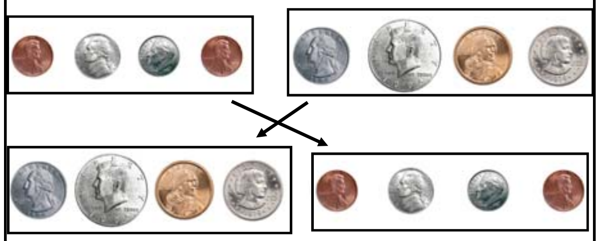
Discovering Algorithms by Manipulating Physical Objects



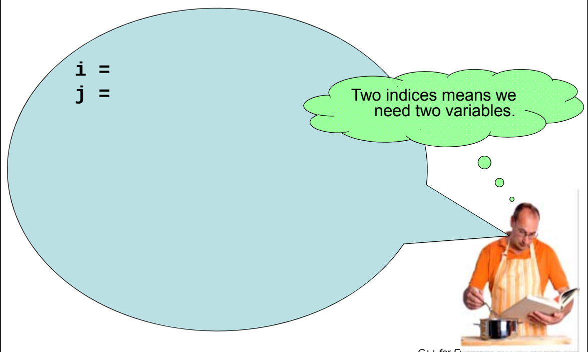
Discovering Algorithms by Manipulating Physical Objects



Discovering Algorithms by Manipulating Physical Objects

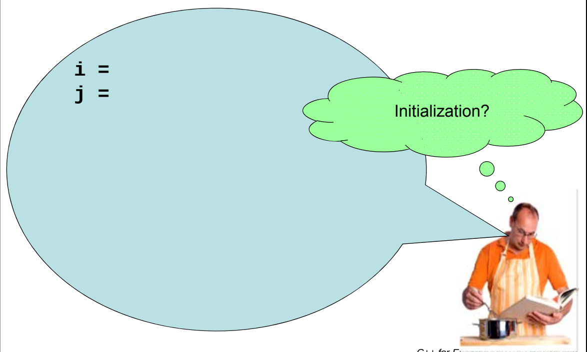


Discovering Algorithms by Manipulating Physical Objects

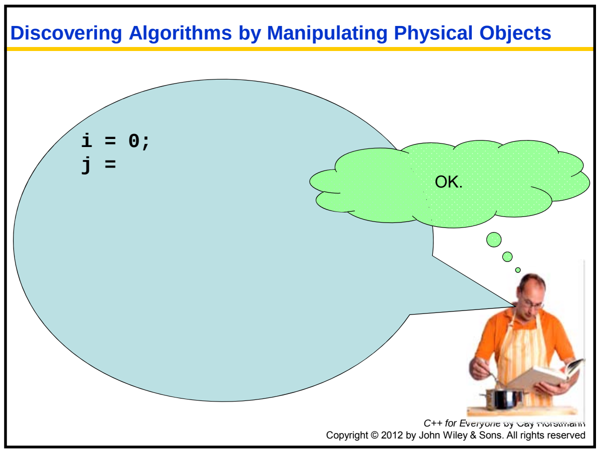


```
i =
j =
```

Discovering Algorithms by Manipulating Physical Objects



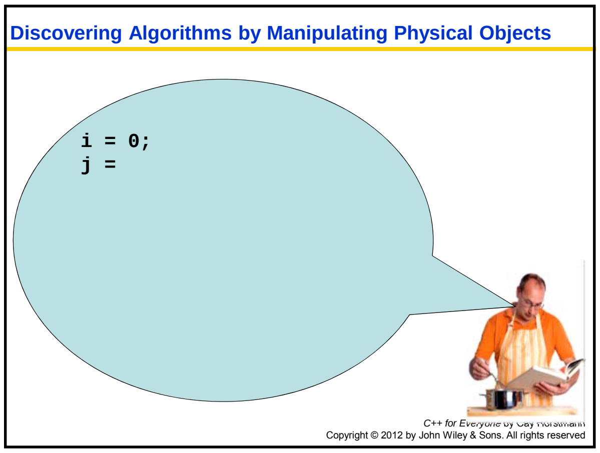
```
i =
j =
```



Discovering Algorithms by Manipulating Physical Objects

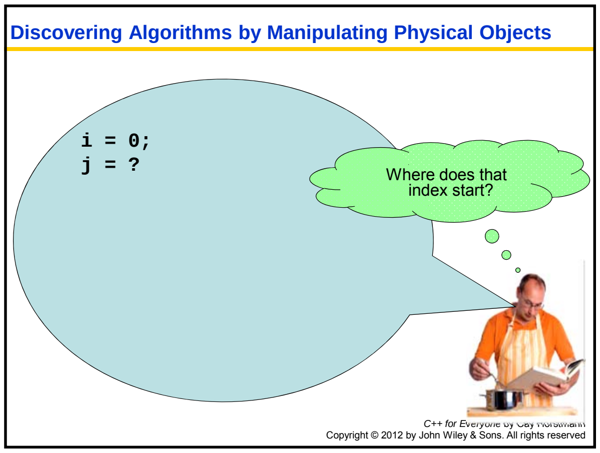
```
i = 0;
j =
```

OK.



Discovering Algorithms by Manipulating Physical Objects

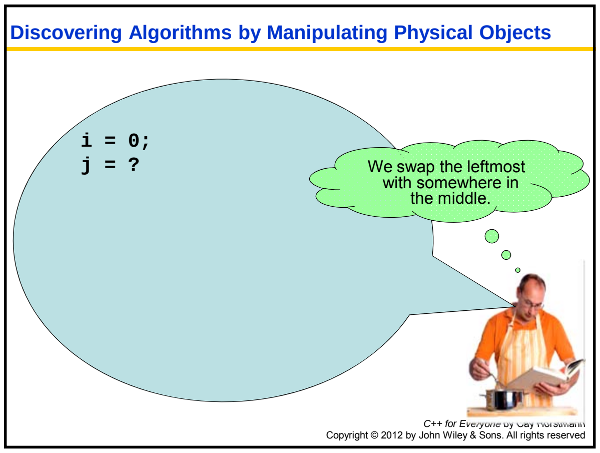
```
i = 0;
j =
```



Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = ?
```

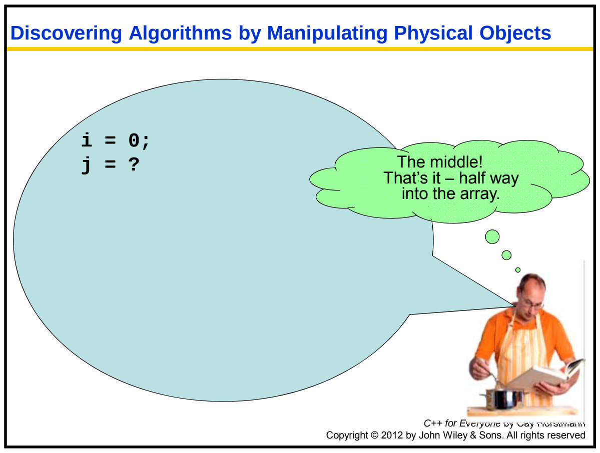
Where does that index start?



Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = ?
```

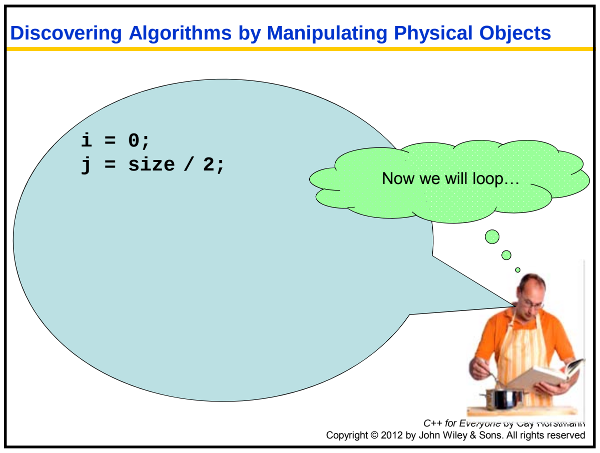
We swap the leftmost with somewhere in the middle.



Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = ?
```

The middle!
That's it – half way into the array.



Discovering Algorithms by Manipulating Physical Objects

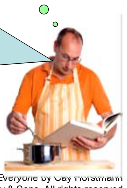
```
i = 0;
j = size / 2;
```

Now we will loop…

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
```

…but…
for how long?
until when?



Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
```

Let's think
about that later.

Discovering Algorithms by Manipulating Physical Objects

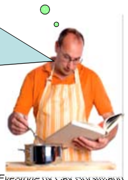
```
i = 0;
j = size / 2;
while ( ??? )
```

For certain we will
be swapping the
elements at the
indices…

Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
   swap elements at i and j
```

…and then go on to
the next pair of
indices to swap…



Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
   swap elements at i and j
   i++;
   j++;
```

But when are we
finished swapping?

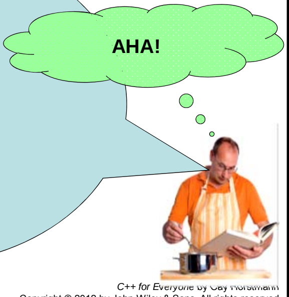
Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
   swap elements at i and j
   i++;
   j++;
```

We only process
half the array

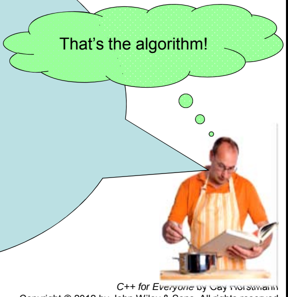
Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while ( ??? )
   swap elements at i and j
   i++;
   j++;
```



Discovering Algorithms by Manipulating Physical Objects

```
i = 0;
j = size / 2;
while (i < size / 2)
   swap elements at i and j
   i++;
   j++;
```



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

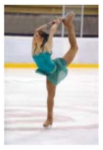
(no, it's not tax time again)

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

Two-Dimensional Arrays

Consider this data from the 2010 Olympic skating competitions:

	Gold	Silver	Bronze
Canada	1	0	1
China1	1	0	
Germany	0	0	1
Korea1	0	0	
Japan0	1	1	
Russia	0	1	1
United States	1	1	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

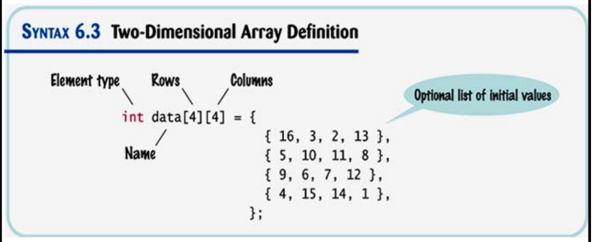
Just as with one-dimensional arrays,
you *cannot* change the size of
a two-dimensional array once it has been defined.

Defining Two-Dimensional Arrays – Initializing

But you can initialize a 2-D array:

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

Defining Two-Dimensional Arrays



Syntax 6.3 Two-Dimensional Array Definition

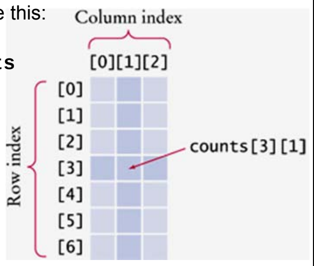
Element type, Rows, Columns, Name, Optional list of initial values

```
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:

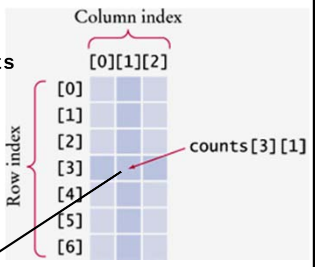
`counts`



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

Defining Two-Dimensional Arrays – Accessing Elements

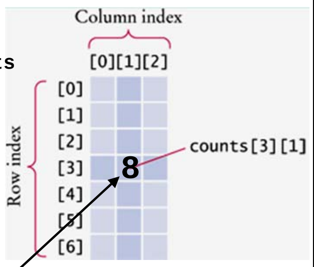
`counts`



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

Defining Two-Dimensional Arrays – Accessing Elements

`counts`



```
// 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;
}
```

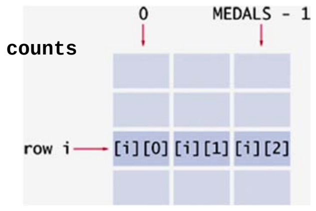
Computing Row and Column Totals

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

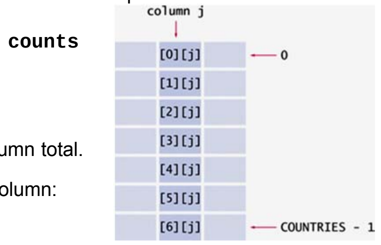
We must be careful to get the right indices.



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 (*metal!*) was won by the set of these particular countries?



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,

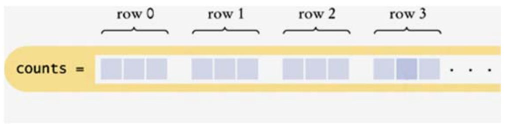
you would have to write
a different function
that has 4 as the parameter.

Hm.

Two-Dimensional Array Parameters

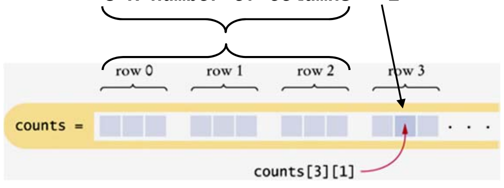
What's the reason behind this?

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.

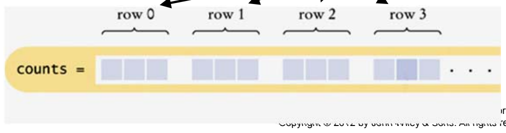
Two-Dimensional Array Parameters

```
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.

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 to build it. and a function must be told about the number elements and possibly the capacity.

It cannot hold more than it's initial capacity.

Arrays – One Drawback

Wouldn't it be good if there were
something that never filled up?

Vectors

A ***vector***

is not fixed in size when it is created

and

it does not have the limitation
of needing an auxiliary variable

AND

you can keep putting things into it

forever!

Well, conceptually forever.
(There's only so much RAM.)

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 *only* **`double`**s

Defining Vectors

By default, a vector is empty when created.

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

Defining Vectors

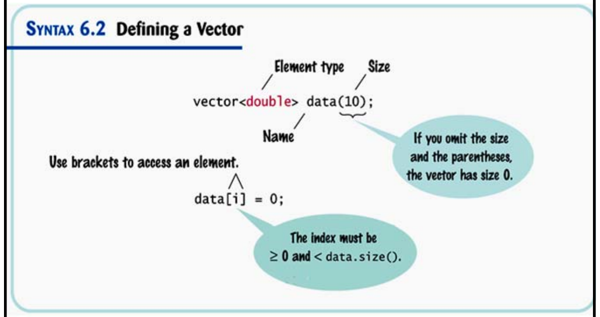
You can specify the initial size.
You still must specify the type of the elements.

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
Element type
Size
Name
If you omit the size and the parentheses, the vector has size 0.
Use brackets to access an element.
The index must be ≥ 0 and < data.size().

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

data[i] = 0;
```

Defining Vectors

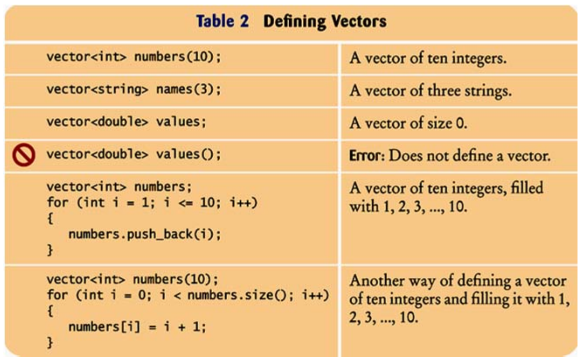


Table 2 Defining Vectors

`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.

Accessing Elements in Vectors

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

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

ERROR!

push_back

So how do you put values into a vector?

You push 'em—

—in the back!

push_back and pop_back

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

```
values.push_back( 32 );
```

push_back and pop_back

```
values.push_back( 32 );
```

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

The vector increases its size by 1.

pop_back

And how do you take them out?

You pop 'em!

—from the back!

push_back and pop_back

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

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

push_back and pop_back

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

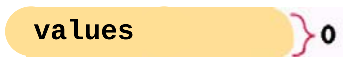
removes the last value from the vector named **values**

and the vector decreases its size by 1.

push_back Adds an Element

```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

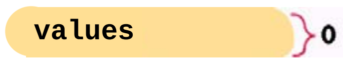
push_back Adds an Element



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

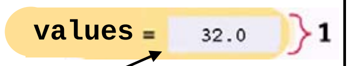
values is an empty vector.
Its size is 0.

push_back Adds an Element



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

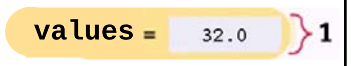
push_back Adds an Element



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

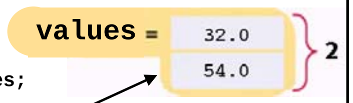
32 is placed into the vector.
Its size is now 1.

push_back Adds an Element



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

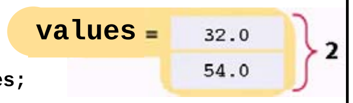
push_back Adds an Element



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

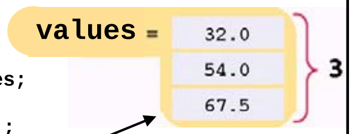
54 is placed into the vector.
It now contains the elements
`32.0` and **`54.0`**,
and its size is **2**.

push_back Adds an Element



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

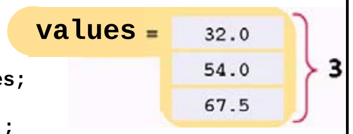
push_back Adds an Element



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

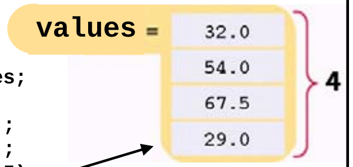
`67.5` is placed into the vector.
It now contains the elements
`32.0`, **`54.0`** and **`67.5`**,
and its size is **3**.

push_back Adds an Element



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

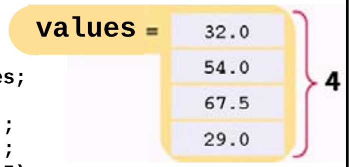
push_back Adds an Element



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

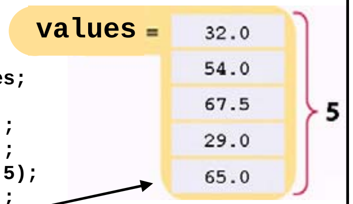
29 is placed into the vector.
It now contains the elements
32.0, **54.0**, **67.5** and **29.0**,
and its size is **4**.

push_back Adds an Element



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

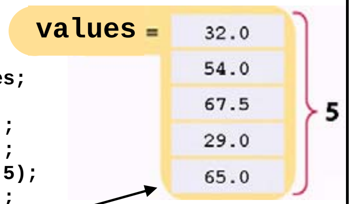
push_back Adds an Element



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

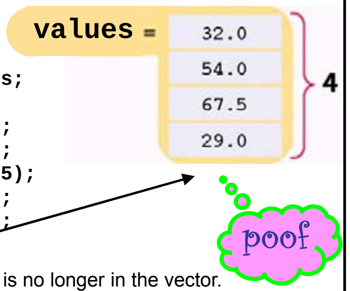
65 is placed into the vector.
It now contains the elements
32.0, **54.0**, **67.5**, **29.0** and **65.0**,
and its size is **5**.

Removing the Last Element with pop_back



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

Removing the Last Element with pop_back



```
vector<double> values;

values.push_back(32);
values.push_back(54);
values.push_back(67.5);
values.push_back(29);
values.push_back(65);
values.pop_back();
```

65 is no longer in the vector.
It now contains only the elements
32.0, **54.0**, **67.5** and **29.0**,
and its size is **4**.

push_back and pop_back

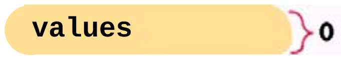
You can use **push_back** to put user input into a vector:

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

push_back Adds an Element

```
vector<double> values;

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

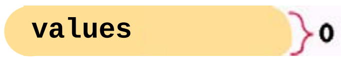
push_back Adds an Element



```
vector<double> values;

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

We are starting again with an empty vector. Its size is 0.

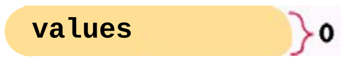
push_back Adds an Element



```
vector<double> values;

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

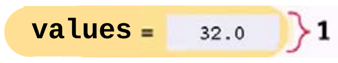
--- The user types **32**

push_back Adds an Element



```
vector<double> values;

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

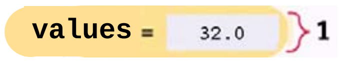
push_back Adds an Element



```
vector<double> values;

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

32 is placed into the vector.
Its size is now 1.

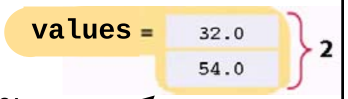
push_back Adds an Element



```
vector<double> values;

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

--- The user types **54**

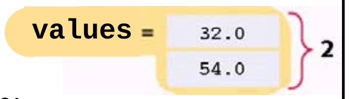
push_back Adds an Element



```
vector<double> values;

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

54 is placed into the vector.
Its size is now 2.

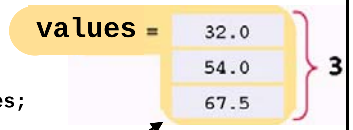
push_back Adds an Element



```
vector<double> values;

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

--- The user types **67.5**

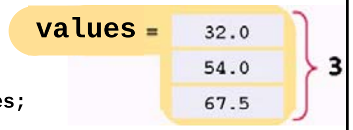
push_back Adds an Element



```
vector<double> values;

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

67.4 is placed into the vector.
Its size is now 3.

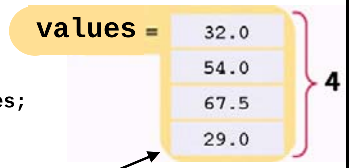
push_back Adds an Element



```
vector<double> values;

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

--- The user types **29**

push_back Adds an Element



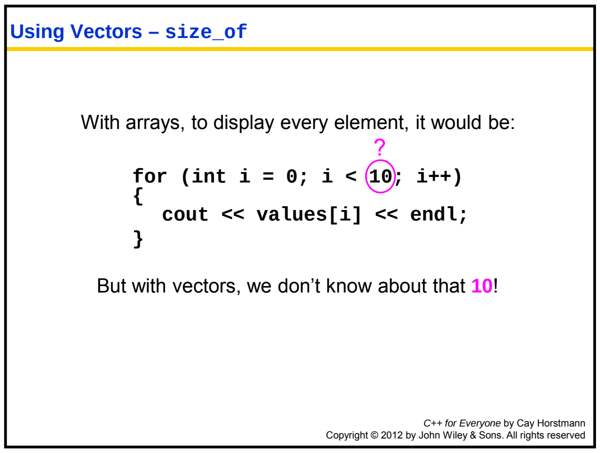
```
vector<double> values;

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

29 is placed into the vector.
Its size is now 4.

Using Vectors – `size_of`

How do you visit every element in an vector?

Recall arrays.

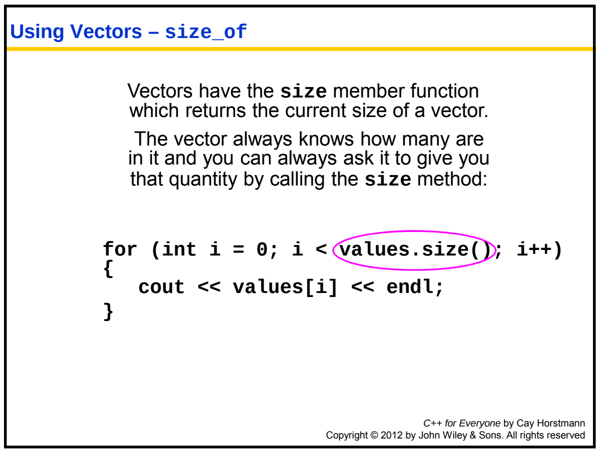


Using Vectors – `size_of`

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

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

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

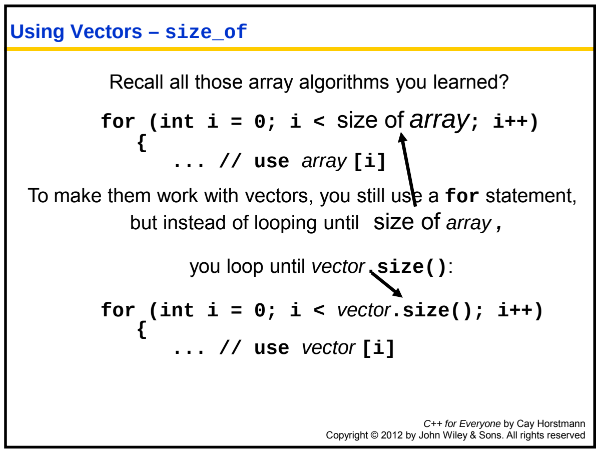


Using Vectors – `size_of`

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

The vector always knows how many 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;
}
```



Using Vectors – `size_of`

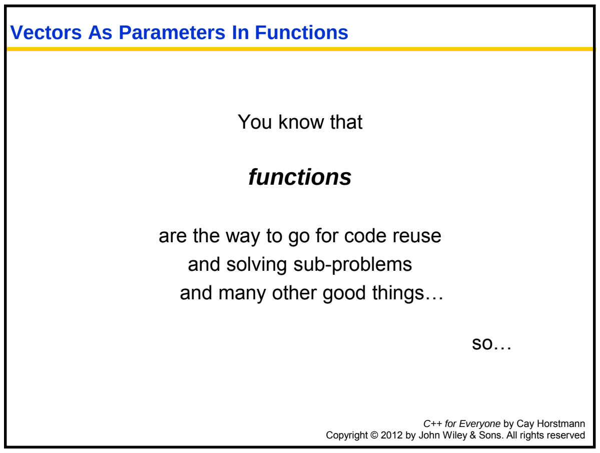
Recall all those array algorithms you learned?

```
for (int i = 0; i < size of array; i++)
   {
      ... // use array[i]
```

To make them work with vectors, you still use a **`for`** statement, but instead of looping until size of *array*,

you loop until *vector*.**`size()`**:

```
for (int i = 0; i < vector.size(); i++)
   {
      ... // use vector[i]
```



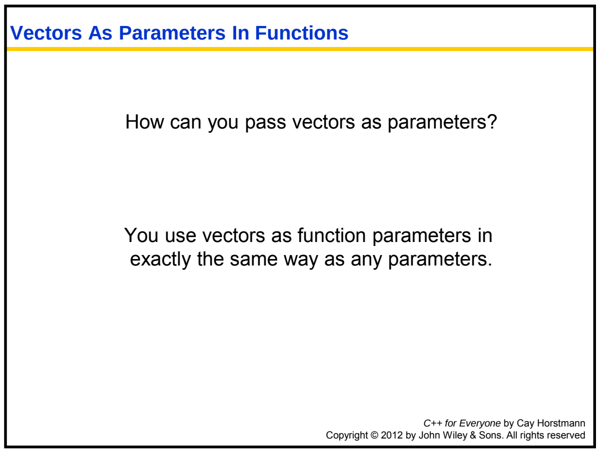
Vectors As Parameters In Functions

You know that

functions

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

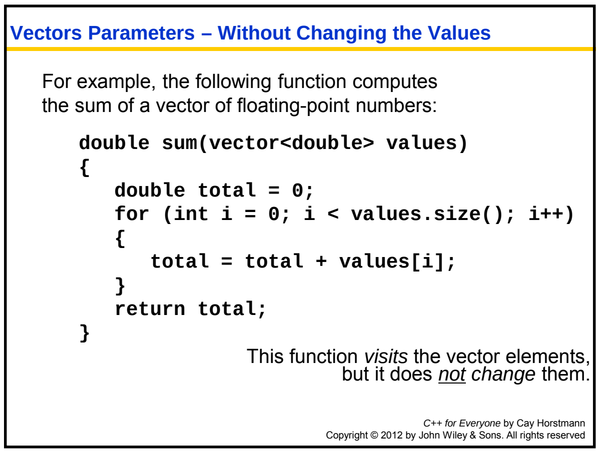
so…



Vectors As Parameters In Functions

How can you pass vectors as parameters?

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



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> 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 it does *not change* them.

Vectors Parameters – Changing the Values

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

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

Vectors Parameters – Changing the Values

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

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

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

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.

Vectors and Arrays as Parameters in Functions

Vectors as parameters are easy.

Arrays are not *quite* so easy.

(vectors… vectors…)

Common Algorithms – Copying, Arrays Cannot Be Assigned

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];
}
```

Common Algorithms – Copying, Vectors Can Be Assigned

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
```

Common Algorithms – Copying, Vectors Can Be Assigned

You can assign a vector to another vector.

Of course they have to hold the same *type* to do this.

```
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
```

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

Common 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,

overwrite the element at that position with the last element in the vector,

then be sure to remove the last element, 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
```

Common 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();
```

Common Algorithms – Inserting an Element, Unordered

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

…oh, this is going to be so easy:

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

Common Algorithms – Inserting an Element, Ordered

However when the elements in a vector are in some order, it's a bit more complicated, just like it was in the array version.

Of course you must know the position, say **pos**, where you will insert the new element.

As in the array version, you need to move all the elements "up".

```
for (int i = last_pos; i > pos; i--)
{
   values[i] = values[i - 1];
}
```

WAIT!!!

Common Algorithms – Inserting an Element, Ordered

You can't do that!

In a vector you cannot assign to the position after the last one!

You cannot assign to any position bigger than `values() – 1`.

OH DEAR!!!

Common Algorithms – Inserting an Element, Ordered

Somehow you need to make
the vector one bigger

before you do the moving.

Common Algorithms – Inserting an Element, Ordered

Be clever.

If you **push_back** the last element:

```
int last_pos = values.size() - 1;
values.push_back(values[last_pos]);
```

…but, but…

Common Algorithms – Inserting an Element, Ordered

Yes, it will be in the vector twice,

but why care?

```
int last_pos = values.size() - 1;
values.push_back(values[last_pos]);
```

You will overwrite it by doing the moving.

Common Algorithms – Inserting an Element, Ordered

And, more importantly,
the vector is now one larger after the **push_back**.
Congratulations, it's to safe go ahead and start moving.

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

And don't forget to insert the new element.
That's what you've been trying to do all along!

Common Algorithms – Inserting an Element, Ordered

Ah.

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

Common Algorithms – Inserting an Element, Ordered

But don't be too clever,

if the position to insert the new element
is after the last element…

…oh, this is going to be so easy,
don't do any moving, just put it there:

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

Inserting into an ordered vector means
inserting into the *middle* of the vector!

Sorting with the C++ Library

Recall that you call the **`sort`** function
to do your sorting for you.

This can be used on vectors also.

The syntax for vectors is even more unusual than arrays:

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

Go ahead and use it as you like.
But don't forget to **`#include <algorithm>`**

Arrays or Vectors? That Is the Question

Should you use arrays or vectors?

(you know you want to say vectors…)

Arrays or Vectors? That Is the Question

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

(say vectors, say vectors…)

Arrays or Vectors? That Is the Question

Vectors can grow and shrink.

(grow, shrink - *think:* vectors…)

Arrays or Vectors? That Is the Question

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

No chance of missing auxiliaries.

Vectors are smarter then arrays!

(size matters and vectors know their own - vectors…)

Arrays or Vectors? That Is the Question

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

(syntax, shmyntax – it's easy too with vectors…)

Arrays or Vectors? That Is the Question

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

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

(only a bit? and *older*? why not be current by using vectors…)

Prefer Vectors over Arrays

So:

Prefer Vectors over Arrays

(it's so nice when the moral of the story is: vectors**!!!**)

CHAPTER SUMMARY

Use arrays for collecting values.

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

CHAPTER SUMMARY

Be able to use common array algorithms.

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

Implement functions that process arrays.

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

CHAPTER SUMMARY

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

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

Discover algorithms by manipulating physical objects.

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

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

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

CHAPTER SUMMARY

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.
- Vectors can occur as function arguments and return values.
- Use a reference parameter to modify the contents of a vector.
- A function can return a vector.