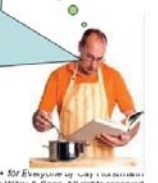
Discovering Algorithms by Manipulating Physical Objects

```
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

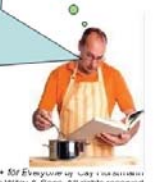
```
i = 0;
j = size / 2;
while ( ??? )
   swap elements at i and j
   i++;
   j++;
```

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++;
```

AHA!



Discovering Algorithms by Manipulating Physical Objects

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

That's the algorithm!

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

Two-Dimensional Arrays 6.6

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

Such data sets commonly occur in financial and scientific applications.

Two-Dimensional Arrays

An arrangement consisting of *tabular data*:

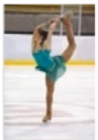
rows and columns of values

is called:

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

Two-Dimensional Arrays

Consider this data from the 2010 Olympic skating competitions:



	Gold	Silver	Bronze
Canada	1	0	1
China	1	1	0
Germany	0	0	1
Korea	1	0	0
Japan	0	1	1
Russia	0	1	1
United States	1	1	0

rows = # countries

columns = 3 (gold, silver, bronze medals)

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.

a two-dimensional array (two subscripts)

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 }
   };
```

row by row

Defining Two-Dimensional Arrays

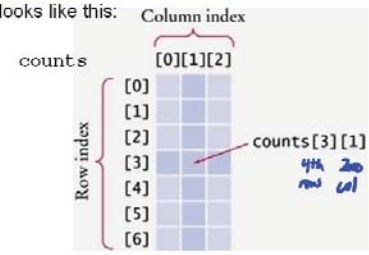
Syntax 6.3 Two-Dimensional Array Definition

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:

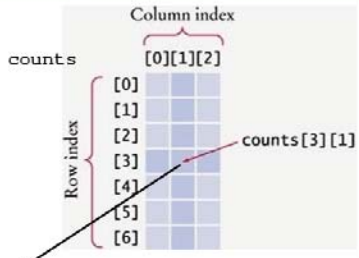


Access to the second element in the fourth row is:

```
counts[3][1]
```

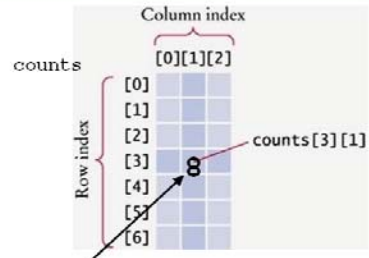
4-1 2-1

Defining Two-Dimensional Arrays – Accessing Elements




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


Defining Two-Dimensional Arrays – Accessing Elements




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


Two-Dimensional Arrays

outer

outer for loop goes through row by row

```
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;
}
```

inner

inner for loop goes through col. by col.

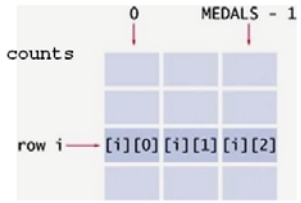
Computing Row and Column Totals

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
```

010
111
101

5 3 0

010 111 101 . . . . .

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,

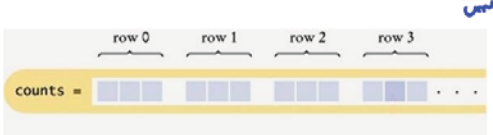
you would have to write
a different function
that has 4 as the parameter.

Hm.

Two-Dimensional Array Parameters

What's the reason behind this?

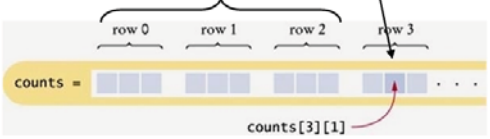
Although the array appears to be two-dimensional,
the elements are still stored as a linear sequence in the computer



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

$$\underbrace{3 \times \textit{number of columns}}_{} + 1$$



Two-Dimensional Array Parameters

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

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

Notice the empty square brackets.

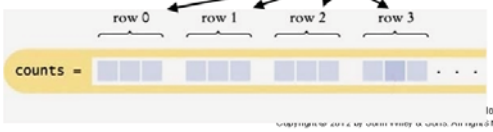
Two-Dimensional Array Parameters

```
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;
```

type name of vector

Note that the element type is enclosed in angle brackets.

`data` can contain *only* `doubles`

Defining Vectors

By default, a vector is empty when created.

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

So data now has 0 elements

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 `doubles` whose initial size is `10`.

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

initial size

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

Defining Vectors

Syntax 6.2 Defining a Vector

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

Element type

Size

Name

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

Use brackets to access an element.

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

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

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 fourth 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 );
```

the dot is necessary

the value you want to add at the end of the vector

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. automatically

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. automatically

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

values } 0

```
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

values } 0

```
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

values = 32.0 } 1

```
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

values = 32.0 } 1

```
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

values = 32.0, 54.0 } 2

```
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

values = 32.0, 54.0 } 2

```
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 =

32.0
54.0
67.5

3

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();
```

values =

32.0
54.0
67.5

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();
```

values =

32.0
54.0
67.5
29.0

4

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();
```

values =

32.0
54.0
67.5
29.0

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();
```

values =

32.0
54.0
67.5
29.0
65.0

5

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();
```

values =

32.0
54.0
67.5
29.0
65.0

5

Removing the Last Element with pop_back

values =

32.0
54.0
67.5
29.0

4

```
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();
```

poof

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

values 0

```
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

values 0

```
vector<double> values;

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

--- The user types 32

push_back Adds an Element

values 0

```
vector<double> values;

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

push_back Adds an Element

values = 32.0 } 1

```
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

values = 32.0 } 1

```
vector<double> values;

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

--- The user types 54

push_back Adds an Element

values = 32.0, 54.0 } 2

```
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

values = 32.0, 54.0 } 2

```
vector<double> values;

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

--- The user types 67.5

push_back Adds an Element

values = 32.0, 54.0, 67.5 } 3

```
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

values = 32.0, 54.0, 67.5 } 3

```
vector<double> values;

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

--- The user types 29

push_back Adds an Element

values =

32.0
54.0
67.5
29.0

4

```
vector<double> values;

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

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

type Q to quit

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;
}
```

? size

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.

You must indicate w/ & when you want a vector to be passed by reference

Vectors Returned from Functions

STOP

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...)