Arrays as Parameters in Functions

There is no `size` member function for arrays.

Arrays as Parameters in Functions

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

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


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.

that's how computer knows it is an array

Arrays as Parameters in Functions

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

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

HEAR YE!
KNOW YE!
THIS BE AN
ARRAY!

AND THIS
BE ITS SIZE

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

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

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

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

Arrays as Parameters in Functions

And writing an ampersand is *always* an error:

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


Arrays as Parameters in Functions

And writing an ampersand is *always* an error:

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


Arrays as Parameters in Functions

And also unlike vectors,

you cannot return an array

Arrays as Parameters in Functions

You cannot return an array.

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


Arrays as Parameters in Functions

The caller must provide an array to be used:

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


A Program Using Vectors as Parameters

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

The program uses three functions:

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

ch06/largest.cpp

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

using namespace std;
```

A Program Using Vectors as Parameters

ch06/largest.cpp

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


A Program Using Vectors as Parameters

ch06/largest.cpp

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


A Program Using Vectors as Parameters

ch06/largest.cpp

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


A Program Using Vectors as Parameters

ch06/largest.cpp

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

   return 0;
}
```


Two-Dimensional Arrays 6.4

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

Such data sets commonly occur in financial and scientific applications.

Two-Dimensional Arrays

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

is called:

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

Two-Dimensional Arrays

Consider this data from the 2006 Olympic skating competitions:

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

Defining Two-Dimensional Arrays

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

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

← 2 sizes

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

Defining Two-Dimensional Arrays – Unchangeable Size

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

Defining Two-Dimensional Arrays – Initializing

But you can initialize a 2-D array:

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

need

Defining Two-Dimensional Arrays – Syntax

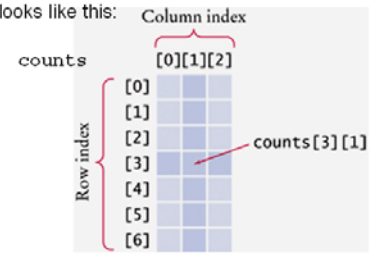
Syntax 6.3 Two-Dimensional Array Definition

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

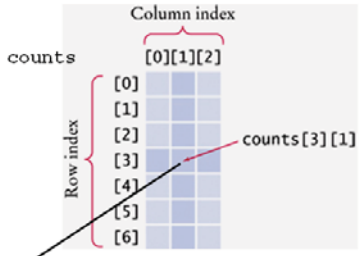
The Olympic array looks like this:



Access to the second element in the fourth row is:

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

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

How to print elements in a 2-dim. array

```
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)`

To total Russia's medal count:

```
i = 3;   // set to Russia row
int total = 0;
for (int j = 0, j < MEDALS, j++)
{
    total = total + counts[i][j];
}
```

Computing Row and Column Totals

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

That would be a column total.

Let j be the silver column:

j=1;

```
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
```

explained in a minute

Two-Dimensional Array Parameters

That function works for only arrays of 3 columns.

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

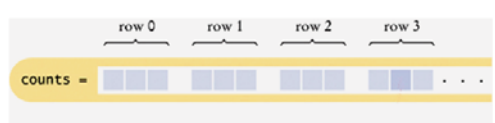
you would have to write
a different function
that has 4 as the parameter.

Hm.

Two-Dimensional Array Parameters

What's the reason behind this?

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



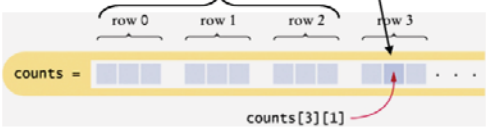
Two-Dimensional Array Parameters

`counts` is stored as a sequence of rows, each 3 long.

So where is `counts[3][1]`?

The offset from the start of the array is

`3 x number of columns + 1`



Two-Dimensional Array Parameters

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

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

Notice the empty square brackets.

Two-Dimensional Array Parameters

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

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

It is!

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

Two-Dimensional Array Parameters

The `row_total` function did not need to know the number of rows of the array.

If the number of rows is required, pass it in:

```
int column_total(int table[][COLUMNS], int rows, int col)
{
   int total = 0;
   for (int i = 0; i < rows; i++)
   {
      total = total + table[i][col];
   }
   return total;
}
```


Two-Dimensional Array Parameters – Common Error

Leaving out the columns value is a very common error.

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

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

Two-Dimensional Array Parameters – Not an Error

Putting a value for the rows is not an error.

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

The compiler just ignores whatever you place there.

Two-Dimensional Array Parameters – Not an Error

Putting a value for the rows is not an error.

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

The compiler just ignores whatever you place there.

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

Never mind

Two-Dimensional Array Parameters

Here is the complete program for medal and column counts.

ch06/medals.cpp

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

using namespace std;

const int COLUMNS = 3;
```


Two-Dimensional Array Parameters

ch06/medals.cpp

```
/**
   Computes the total of a row in a table.
   @param table = a table with 3 columns
   @param row = the row that needs to be totaled
   @return the sum of all elements in the given row
*/
double row_total(int table[][COLUMNS], int row)
{
   int total = 0;
   for (int j = 0; j < COLUMNS; j++)
   {
      total = total + table[row][j];
   }
   return total;
}
```


Two-Dimensional Array Parameters

ch06/medals.cpp

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

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


Two-Dimensional Array Parameters

ch06/medals.cpp

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


Two-Dimensional Array Parameters

ch06/medals.cpp

```
   cout << "    Country  Gold  Silver  Bronze   Total"
      << endl;

   // Print countries, counts, and row totals
   for (int i = 0; i < COUNTRIES; i++)
   {
      cout << setw(15) << countries[i];
      // Process the ith row
      for (int j = 0; j < MEDALS; j++)
      {
         cout << setw(8) << counts[i][j];
      }
      int total = row_total(counts, i);
      cout << setw(8) << total << endl;
   }
   return 0;
}
```

CHAPTER SUMMARY

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

CHAPTER SUMMARY

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