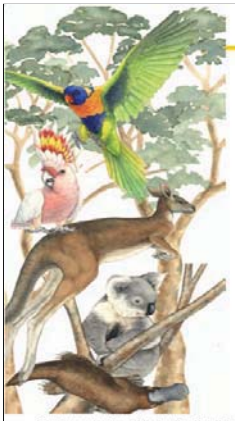


Chapter Six: Arrays and Vectors

Slides prepared by Evan Gallagher, New York University

Chapter Goals

- To become familiar with using arrays and vectors to collect values
- To learn about common algorithms for processing arrays and vectors
- To write functions that receive and return arrays and vectors
- To learn how to use two-dimensional arrays

Using Arrays and Vectors

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

Using Vectors

- When you need to work with a large number of values – all together, the vector construct is your best choice.
- By using a ***vector*** you
  - can conveniently manage collections of data
  - do not worry about the details of how they are stored
  - do not worry about how many are in the vector
    - a vector automatically grows to any desired size

Using Arrays

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

Using Arrays and Vectors

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

Using Arrays and Vectors

Think of a sequence of data:

32 54 67.5 29 35 80 115 44.5 100 65

(all of the same type – real numbers)

Using Arrays and Vectors

largest

32 54 67.5 29 35 80 115 44.5 100 65

Which is the largest in this set?

(You must look at every single value to decide.)

Using Arrays and Vectors

32 54 67.5 29 35 80 115 44.5 100 65

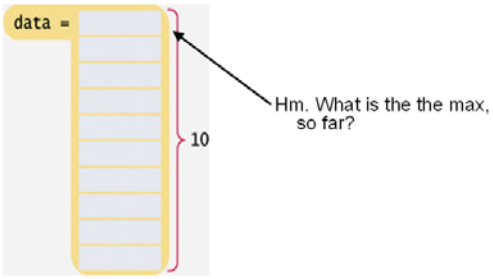
So you would create a variable for each, of course!

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

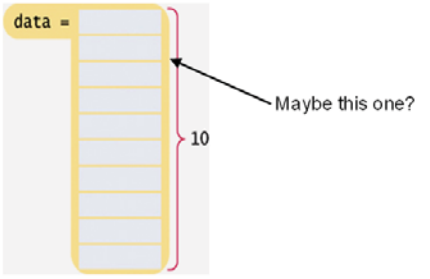
Then what ???

Using Arrays and Vectors

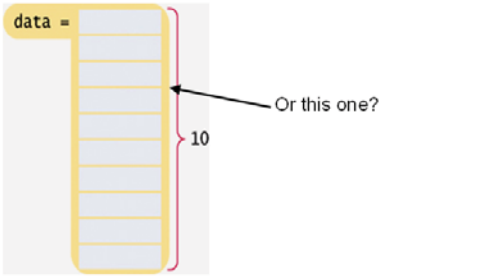
You can easily visit each element in an array, checking and updating a variable holding the current maximum.



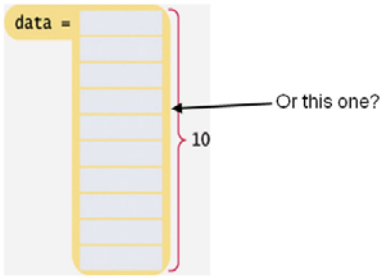
Using Arrays and Vectors



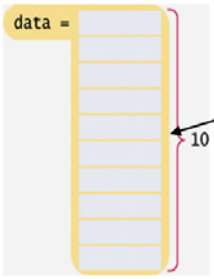
Using Arrays and Vectors



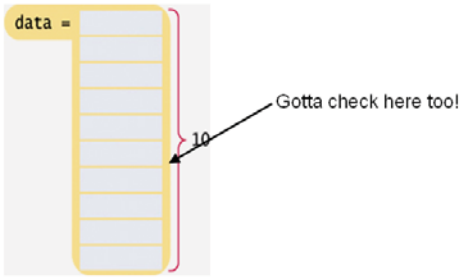
Using Arrays and Vectors



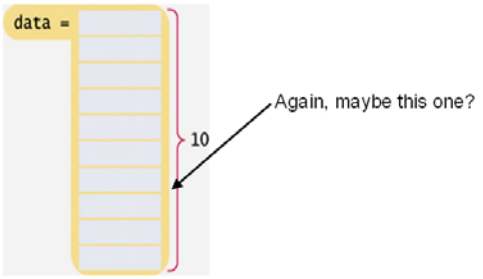
Using Arrays and Vectors



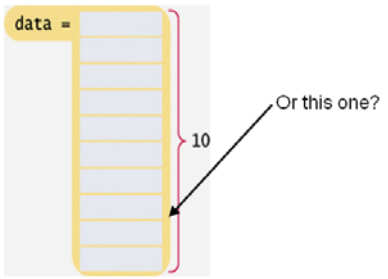
Using Arrays and Vectors



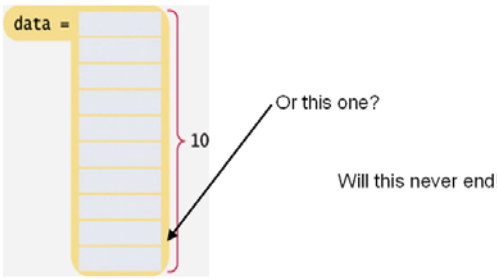
Using Arrays and Vectors



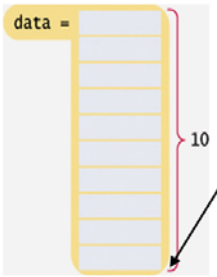
Using Arrays and Vectors



Using Arrays and Vectors



Using Arrays and Vectors



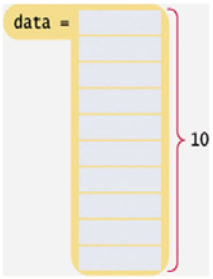
Or the last one? ***Finally!***

Using Arrays and Vectors

That would have been impossible with ten separate variables!

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


Defining Arrays



An "array of double type"

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

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

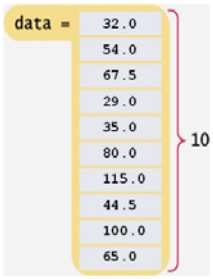
type of each element

quantity of elements – the "size" of the array

Defining Arrays with Initialization

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

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



list elements of an array in curly braces separated by commas

Accessing an Array Element

An array element can be used like any variable.

To access an array element, you use the notation:

```
data[i]
```

where `i` is the *index*.

Computers "number" the elements 0 to size - 1

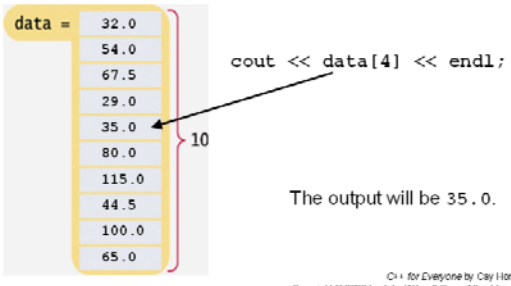
Accessing an Array Element

Put the junk mail in there

in `mailboxes[356]`

Accessing an Array Element

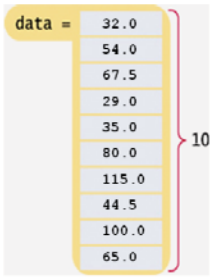
To access the element at index 4 using this notation: data[4]
4 is the *index*.



The output will be `35.0`.

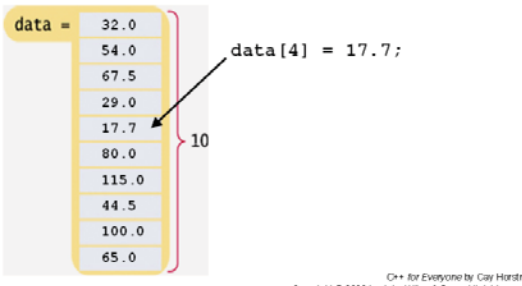
Accessing an Array Element

The same notation can be used to change the element.



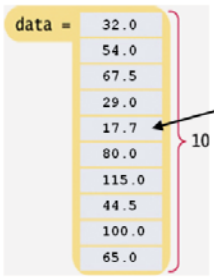
Accessing an Array Element

The same notation can be used to change the element.



Accessing an Array Element

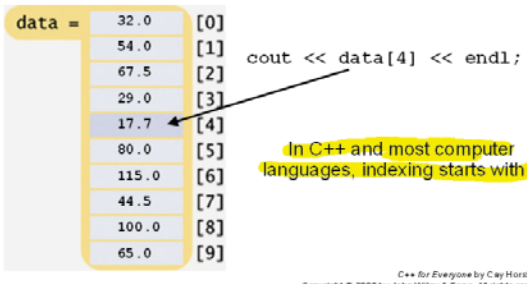
The same notation can be used to change the element.



The output will be `17.7`.

Accessing an Array Element

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



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

Accessing an Array Element

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

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

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

Illegally Accessing an Array Element – *Bounds Error*

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

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

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

DANGER!!!

DANGER!!!

DANGER!!!

Array Syntax

Syntax 6.1 Defining an Array

Size must be a constant.

Ok to omit size if initial values are given.

Element type Name Size

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

Optional list of initial values

Use brackets to access an element.

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

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

or <= size-1

Array Syntax

Table 1 Defining Arrays

`int numbers[10];`	An array of ten integers.
`const int SIZE = 10;` `int numbers[SIZE];`	It is a good idea to use a named constant for the size.
`int size = 10;` `int numbers[size];`	**Error:** The size must be a constant.
`int squares[5] = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`int squares[] = { 0, 1, 4, 9, 16 };`	You can omit the array size if you supply initial values. The size is set to the number of initial values.
`int squares[5] = { 0, 1, 4 };`	If you supply fewer initial values than the size, the remaining values are set to 0. This array contains 0, 1, 4, 0, 0.
`string names[3];`	An array of three strings.

Using Arrays – Visiting All Elements

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

A `for` loop's variable is best:

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

This example will print each element of the array "data" on a separate line.

Using Arrays – Visiting All Elements

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

A `for` loop's variable is best:

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

When `i` is `0`,

Using Arrays – Visiting All Elements

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

A `for` loop's variable is best:

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

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

Using Arrays – Visiting All Elements

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

A `for` loop's variable is best:

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

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

When `i` is `1`,

Using Arrays – Visiting All Elements

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

A `for` loop's variable is best:

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

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

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

Using Arrays – Visiting All Elements

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

A `for` loop's variable is best:

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

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

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

When `i` is `2`,

Using Arrays – Visiting All Elements

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

A `for` loop's variable is best:

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

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

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

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

Using Arrays – Visiting All Elements

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

A `for` loop's variable is best:

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

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

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

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

...

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

Using Arrays – Visiting All Elements

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

A `for` loop's variable is best:

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

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

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

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

...

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

Using Arrays – Visiting All Elements

To visit all elements of an array, use a variable for the index.
A `for` loop's variable is best:

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

Note that the loop condition is that the index is

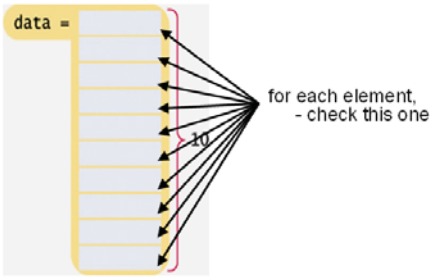
***less than* 10**

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

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

Using Arrays and Vectors

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



Arrays – One Drawback

The size of an array *cannot* be changed after it is created.

You have to get the size right – *before* you define an array.
The compiler has to know the size.

What is the size?

That can be a hard question sometimes!

Vectors

Vectors to the rescue!

Vectors

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

just like the array does,

but its size can change.

Defining Vectors

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

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

Defining Vectors

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

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

Note that the element type is enclosed in angle brackets.

`data` can contain `doubles`

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 `doubles` whose initial size is `10`.

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

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

note size is enclosed in parentheses NOT square brackets

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().

go back to square brackets when accessing elements of a vector

Accessing Elements in Vectors

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

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

HOWEVER...

Accessing Elements in Vectors

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

EMPTY!

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

ERROR!

push_back

So how do you put values into a vector?

You stuff them in—

— at the end!

pop_back

And how do you take them out?

You pop 'em!

push_back and pop_back

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

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

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

gets added to the end of vector data

push_back and pop_back

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

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

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

removes a value from the vector named `data`.

push_back Adds an Element

data } 0

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

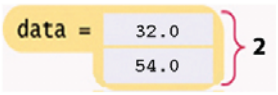

push_back Adds an Element

data = 32.0 } 1

```
vector<double> data;

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

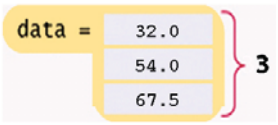
push_back Adds an Element




```
vector<double> data;

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

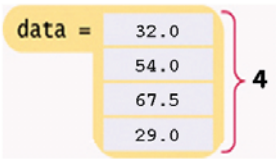

push_back Adds an Element




```
vector<double> data;

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

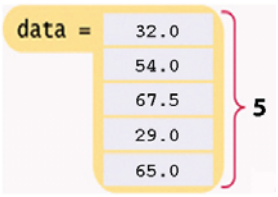

push_back Adds an Element




```
vector<double> data;

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

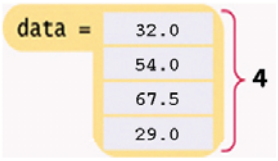

push_back Adds an Element




```
vector<double> data;

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


Removing the Last Element with pop_back




```
vector<double> data;

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


push_back and pop_back

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

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

how to have user enter values into a vector

push_back Adds an Element

data } 0

```
vector<double> data;

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

We are starting again with an empty vector

push_back Adds an Element

data } 0

```
vector<double> data;

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

The user types 32

push_back Adds an Element

data = 32.0 } 1

```
vector<double> data;

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

32 is placed into the vector

push_back Adds an Element

data = 32.0 } 1

```
vector<double> data;

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

The user types 54

push_back Adds an Element

data = 32.0, 54.0 } 2

```
vector<double> data;

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

54 is placed into the vector

push_back Adds an Element

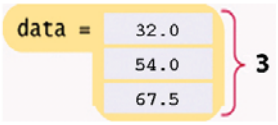
data = 32.0, 54.0 } 2

```
vector<double> data;

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

The user types 67.5

push_back Adds an Element

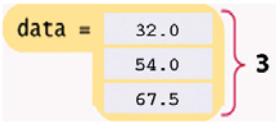



```
vector<double> data;

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

67.5 is placed into the vector

push_back Adds an Element

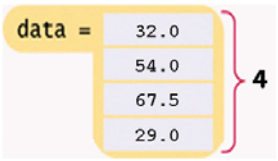



```
vector<double> data;

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

The user types 29

push_back Adds an Element




```
vector<double> data;

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

29 is placed into the vector

when user types q (for quit) the loop will terminate b/c cin >> input will be false

Defining Vectors

Table 2 Defining Vectors

`vector<int> numbers(10);`	A vector of ten integers.
`vector<string> names(3);`	A vector of three strings.
`vector<double> values;`	A vector of size 0.
`vector<double> values();`	**Error:** Does not define a vector.

`vector<int> numbers;`
`for (int i = 1; i <= 10; i++)`
`{`
`  numbers.push_back(i);`
`}`
	A vector of ten integers, filled with 1, 2, 3, ..., 10.

`vector<int> numbers(10);`
`for (int i = 0; i < numbers.size(); i++)`
`{`
`  numbers[i] = i + 1;`
`}`
	Another way of defining a vector of ten integers and filling it with 1, 2, 3, ..., 10.

Using Vectors – Visiting Every Element

How do you visit every element in an vector?

Using Vectors – Visiting Every Element

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

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

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

Using Vectors – Visiting Every Element

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

better to use

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


Partially-Filled Arrays

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

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

So it's the same question as before:

What is the size?

Partially-Filled Arrays – Capacity

What is the size?

We guess.

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

We call this quantity the *capacity*.

Partially-Filled Arrays – Capacity

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

We would set the capacity with a `const`:

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


Partially-Filled Arrays

This array will usually have less than CAPACITY elements in it

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

Partially-Filled Arrays – Companion Variable for Size

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

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

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

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

Partially-Filled Arrays – Capacity

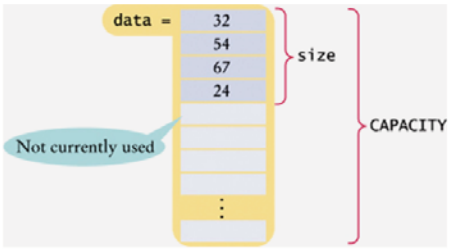
Whenever the size of the array changes we update this variable:

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

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


Partially-Filled Arrays – Companion Variable for Size

If only four elements have been stored in the array:



Partially-Filled Arrays – Capacity

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

By using the `size` companion variable.

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


Arrays Cannot Be Assigned, Vectors Can STOP

Suppose you have two arrays

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

The following assignment is an error:

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

You must use a loop to copy all elements:

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


Arrays Cannot Be Assigned, Vectors Can

Vectors do not suffer from this limitation.

Consider this example:

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


Arrays or Vectors? That Is the Question

Should you use arrays or vectors?

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

Vectors can grow and shrink.

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

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

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