Arrays

The Range-Based for Loop

- C++ has a fourth loop type (after while, do-while, and for)
- its official name is the range-based for loop, but everyone calls it the foreach loop
- it is extremely common to need to access each element of an array, one by one, in order, from beginning to end

The Range-Based for Loop

- C++ has a fourth loop type (after while, do-while, and for)
- its official name is the range-based for loop, but everyone calls it the foreach loop
- it is extremely common to need to access each element of an array, one by one, in order, from beginning to end

- you can do this with a while, a do-while, or a for loop, e.g.:

```
int values[]= {10, 20, 30, 40, 50};
for (unsigned index = 0; index < 5; index++)
{
   ... do something with values[index]
}
```

The Range-Based for Loop

- but this construct is so common that there is a special way of doing exactly this: the foreach loop

```
int values[] = {10, 20, 30, 40, 50};

for (int v : values)
{
   ... do something with value
}
```

- you do not have to specify the starting and ending indices
- you do not have to increment an index
- you do not have to use brackets
- the foreach loop gives you each element directly, one at a time

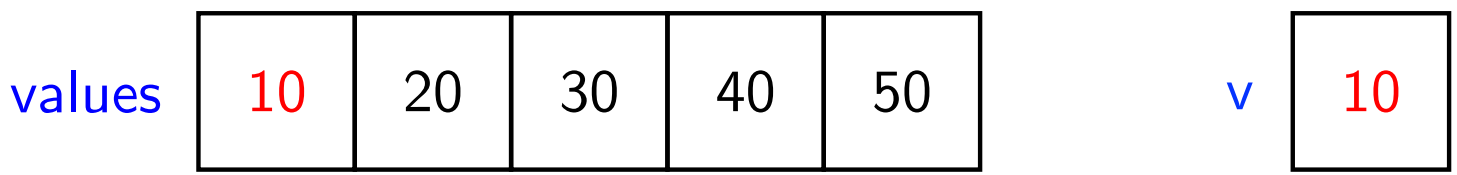
foreach Under the Hood

```
unsigned values[] ={10, 20, 30, 40, 50};
for (unsigned v : values) {

  // do something with value

}
```

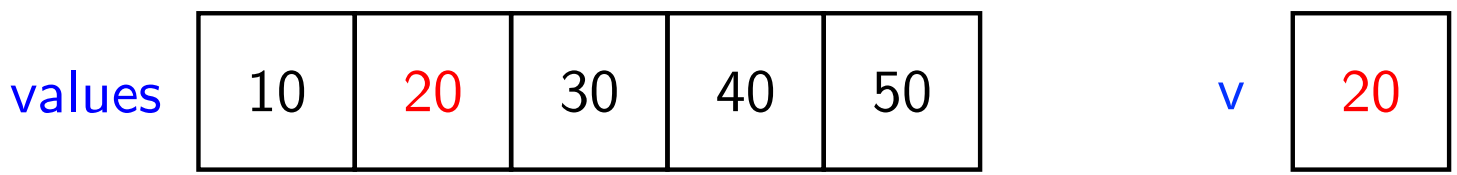
- v is a separate variable
- each time through the loop, it gets a copy of the next element of the array



foreach Under the Hood

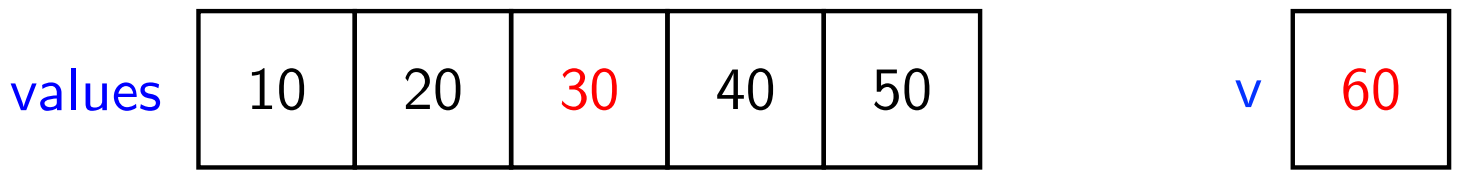
```
unsigned values[]= {10, 20, 30, 40, 50};
for (unsigned v : values)
{
  // do something with value
}
```

- v is a separate variable
- each time through the loop, it gets a copy of the next element of the array



foreach Under the Hood

- v is a separate variable
- each time through the loop, it gets a copy of the next element of the array
- you can do anything you wish with that value
  - output it
  - calculate with it
  - use it as the argument of a function call
  - even change it by assigning a different value to it
- but it is a copy of what is in the array
- anything you do to value has no effect on the array element it was copied from e.g.: `value *= 2;`



foreach Under the Hood

- many times this is what you want
- the array is storing data
- the foreach loop lets you access that data without accidentally changing it
- but sometimes you really do need to modify the values in the array

foreach Under the Hood

- many times this is what you want
- the array is storing data
- the foreach loop lets you access that data without accidentally changing it
- but sometimes you really do need to modify the values in the array

- suppose you need to add 10 to every element in an array
- this, of course, won't work:

```
unsigned values[]= {10, 20, 30, 40, 50};

for (unsigned val : values)
{
  val += 10;
}
```

foreach Under the Hood

- to modify the values in the array using a foreach loop you must use a reference variable instead of a normal (copy) variable

```
unsigned values[]= {10, 20, 30, 40, 50};

for (unsigned & val : values)
{
  val += 10;
}
```

- now val is not a copy of the array element
- it is an alias of the array element
- a change made to val is actually being made to the array element itself

see program_7_12.cpp

foreach vs. for

- use the foreach loop to access array elements when you only need the elements themselves
- use the for loop to access array elements when you want to use the indices of the array elements

```
for (int val : values) {
  cout << val << endl; }
```

```
10
20
30
40
50
```

```
for (unsigned index = 0; index < SIZE;
     index++)
{
  cout << "the element at index " <<
    index << " is " << values[index] << endl;
}
```

```
the value at index 0 is 10
the value at index 1 is 20
the value at index 2 is 30
the value at index 3 is 40
the value at index 4 is 50
```

Whole-Array Assignment

- I would like to copy an entire array's values to another array

```
const unsigned SIZE = 4;
int array1[] = {-2, -1, 0, 1};
int array2[SIZE];

array2 = array1;
```

Whole-Array Assignment

- I would like to copy an entire array's values to another array

```
const unsigned SIZE = 4;
int array1[] = {-2, -1, 0, 1};
int array2[SIZE];

array2 = array1;
```

- this will not work!
- remember: the name array1 refers to the address of the first byte of the first element of array1
- `array2 = array1;` is interpreted as, "change the place where array2's first byte is to the same place where array1's first byte is"
- but we cannot move the place where a variable is located in memory to a different place

Whole-Array Assignment

- the only way to copy an array's values to another array is element-by-element

```
for (unsigned count = 0; count < SIZE; count++)
{
  array2[count] = array1[count];
}
```

Whole-Array Comparison

- I would like to verify if two arrays have the same elements

```
int array1[] {-2, -1, 0, 1};
int array2[] {-2, -1, 0, 1};

  array1 == array2 // should this be true?
```

Whole-Array Comparison

- We would like to see if two arrays have the same elements `int array1[] {-2, -1, 0, 1};`
`int array2[] {-2, -1, 0, 1};`

```
array1 == array2 // should be true?
```

- this will not work!
- remember, array1 is really an address (say, 8610)
- and array2 is a different address (say, 8626)
- array1 == array2 really means 8610 == 8626, which is clearly false

Whole-Array Comparison

- the only way to compare an array's values to another array is element-by-element
- what would the code for this look like?

Whole-Array Comparison

- the only way to compare an array's values to another array is element-by-element
- what would the code for this look like?

```
const int size = 5;
bool same = true;
for(unsigned index = 0;index<SIZE;index++)
{
   same = array1[index] == array2[index];
   if(same == false) break;
}


if(same == true)
        cout<<"Two arrays are the same"<<endl;
else
        cout<<"Two arrays are NOT the same"<<endl;
```

Common Array Algorithms

- all of the following common algorithms use the foreach loop
- this is correct if every element has a value
- if not, you must use an index — see below, partially filled arrays

Common Array Algorithms

- all of the following common algorithms use the foreach loop
- this is correct if every element has a value
- if not, you must use an index — see below, partially filled arrays

- print the contents

```
for (int item : items) {
  cout << item << endl; }
```

- sum the contents

```
unsigned total = 0;
for (int value : values) {
  total += value;
}
```

Common Array Algorithms

- compute the average

```
double total = 0.0;
for (int value : values)
{
  total += value;
}
double average = static_cast<float>(total) / NUMBER_OF_VALUES;
```

Common Array Algorithms

- find the largest value

```
int largest = MIN_VALUE;
for (int value : values)
{
  if (value > largest)
  {
    largest = value;
  }
}
```

Common Array Algorithms

- find the position of the largest value
- remember, if you need the index, then for loop is better

```
int largest = MIN_VALUE;
unsigned position_of_largest = 0;
for (unsigned index = 0; index < SIZE; index++)
{
  if (values[index] > largest)
  {
    largest = values[index];
    position_of_largest = index;
  }
}
```

Omitted

- we will not discuss Partial Array Initialization on pages 391–392