Arrays

Class 24

Variable Size

- all the variables we have declared so far are exactly large enough for one value of the declared type

```
int value;
```

int 1234 value

```
double price;
```

double 123.4567 price

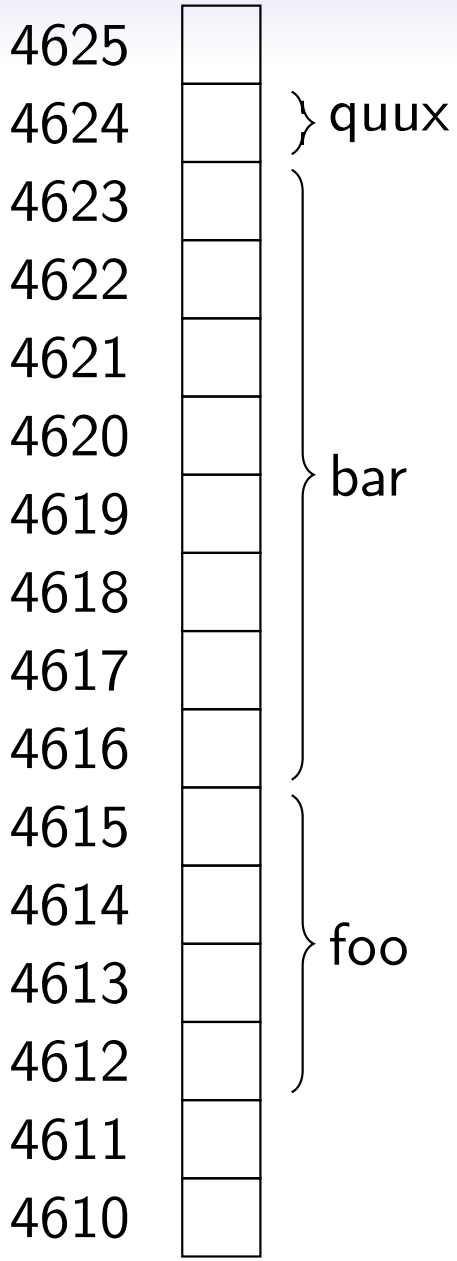
```
char initial;
```

char 'A' initial

Variables in Memory

- a computer's memory is a list of numbered locations, each of which refers to a byte of 8 bits
- the number of a byte is its address
- a simple variable (e.g., unsigned or double) refers to a location of memory containing a number of consecutive bytes
- the number of bytes is determined by the type of the variable (e.g., 4 bytes for unsigned, 8 bytes for double)
- the address of the variable is the address of the first byte of memory where it is stored

```
int main()
{
 unsigned foo; // address 4612
 double bar; // address 4616
 bool quux; // address 4624
}
```



Array Variable

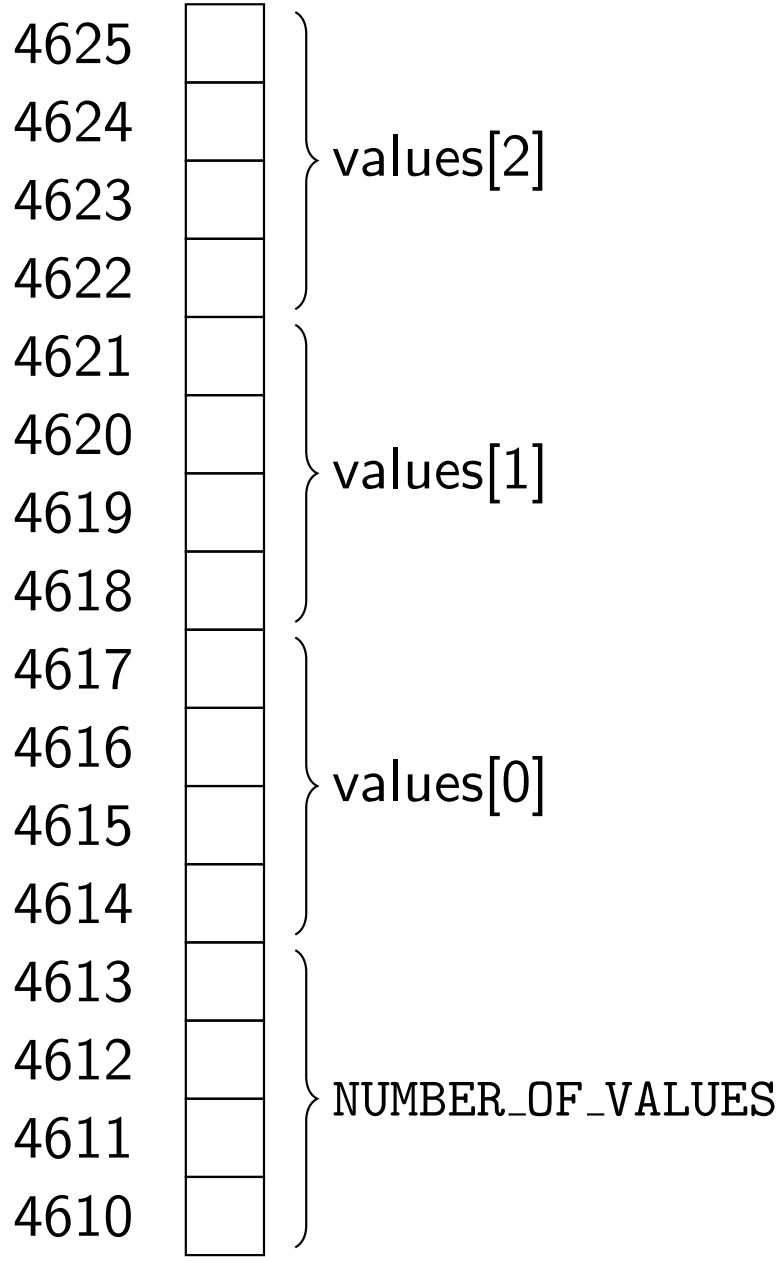
- an array acts like a variable that can store many values
  - all of the same type
  - contiguously, one after the other, in memory

```
const unsigned NUMBER_OF_VALUES = 3;
int values[NUMBER_OF_VALUES];
```

- allocates enough memory to hold three integers

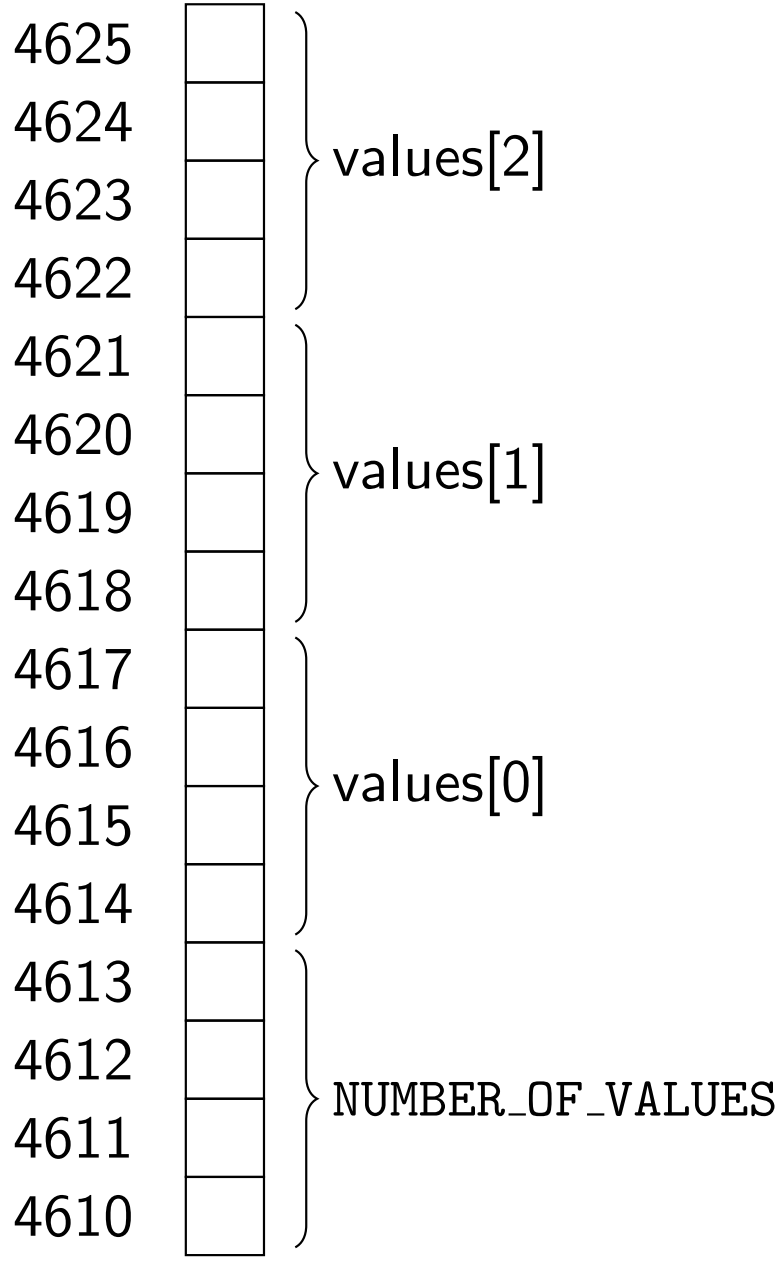
```
const unsigned NUMBER_OF_VALUES = 3;
int values[NUMBER_OF_VALUES];
```

- the address of `NUMBER_OF_VALUES` is 4610
- the address of `values[2]` is 4622



```
const unsigned NUMBER_OF_VALUES = 3;
int values[NUMBER_OF_VALUES];
```

- the address of `NUMBER_OF_VALUES` is 4610
- the address of `values[2]` is 4622
- the address of values is 4614, the same as `values[0]`
- both values and values[0] refer to the same location
- but values[0] means the contents of one element of the array, while values is a synonym for 4614



The Value of the Array Variable Itself

```
int main()
{
  int values[] {10, 20, 30};

  cout << values[0] << endl;
  cout << values << endl;
}
```

- to emphasize, when run on borax, the literal output is:

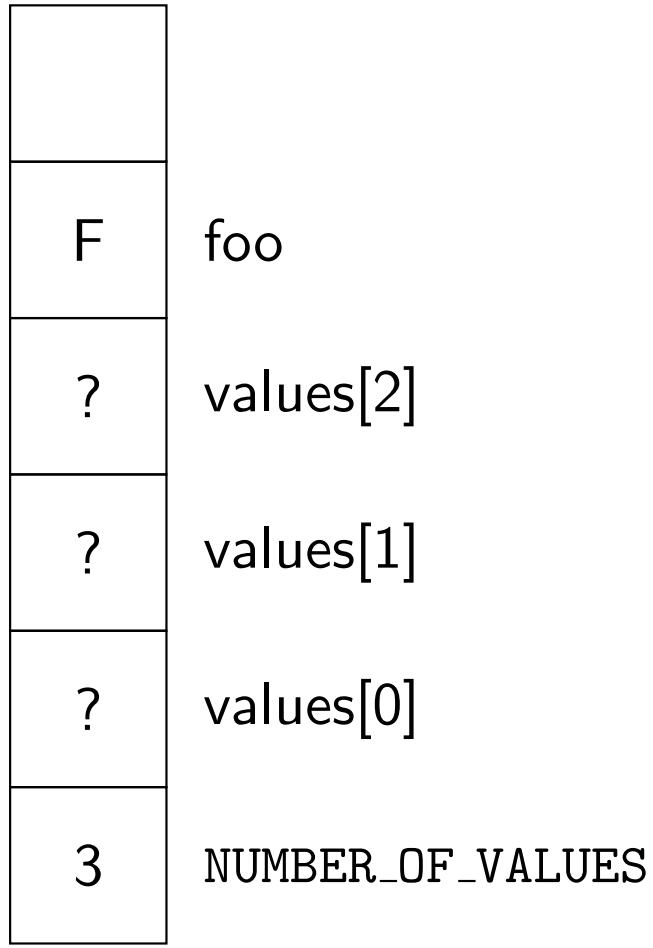
```
10
0x7ffd947d5d40
```

- the latter being the actual physical address in hexadecimal of the location in memory where values is stored

```
const unsigned NUMBER_OF_VALUES = 3;
int values[NUMBER_OF_VALUES];
bool foo = false;
```

- normally, however, we do not show the individual bytes of a variable
- or even the exact address values

- there is no `values[3]` but if there were, it would be above `values[2]` where foo is



Arrays

```
double temperatures[100]; // can hold 100 doubles
string names[50];         // can hold 50 strings
unsigned counts[500];     // can hold 500 unsigned ints
```

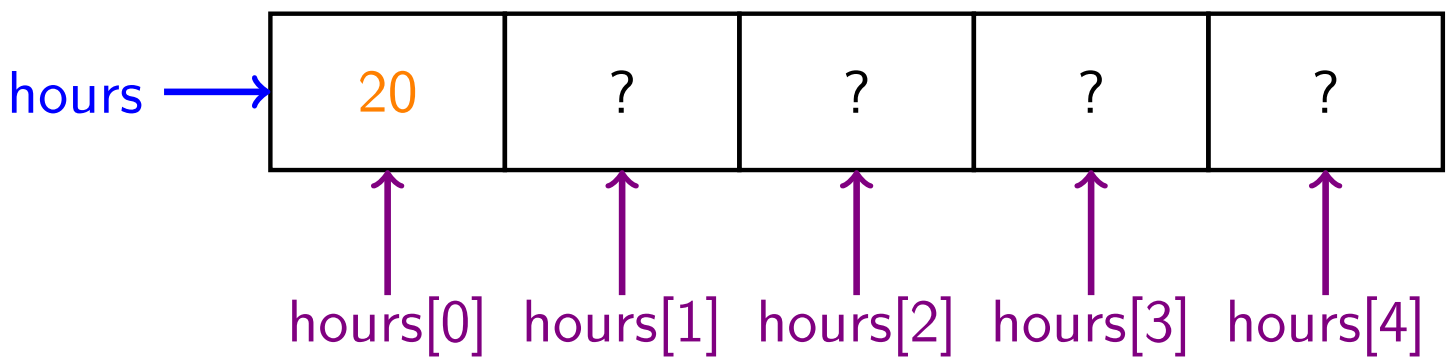
- the amount of RAM used by an array is exactly the number of bytes for one element times the number of elements
- `double temperatures[1000];` on ice would consume

$$8 \text{ bytes per double} \times 1000 \text{ doubles} = 8000 \text{ bytes}$$

Array Elements

- the entire array has one name
- individual elements can be accessed using subscripts
- every element in every array is numbered
- the numbers always start at 0 and go up, so they are always unsigned integers
- a subscript is an unsigned integer expression in square brackets following the name

```
unsigned hours[5];
hours[0] = 20;
```



Arrays

- there are two different types associated with an array
  1. the index type: since indices start at 0 and go up, the index type is always an unsigned integer type
  2. the element type: this can be any type, e.g., int, double, string, unsigned
- do not confuse the two

Arrays

- there are two different types associated with an array
  1. the index type: since indices start at 0 and go up, the index type is always an unsigned integer type
  2. the element type: this can be any type, e.g., int, double, string, unsigned
- do not confuse the two

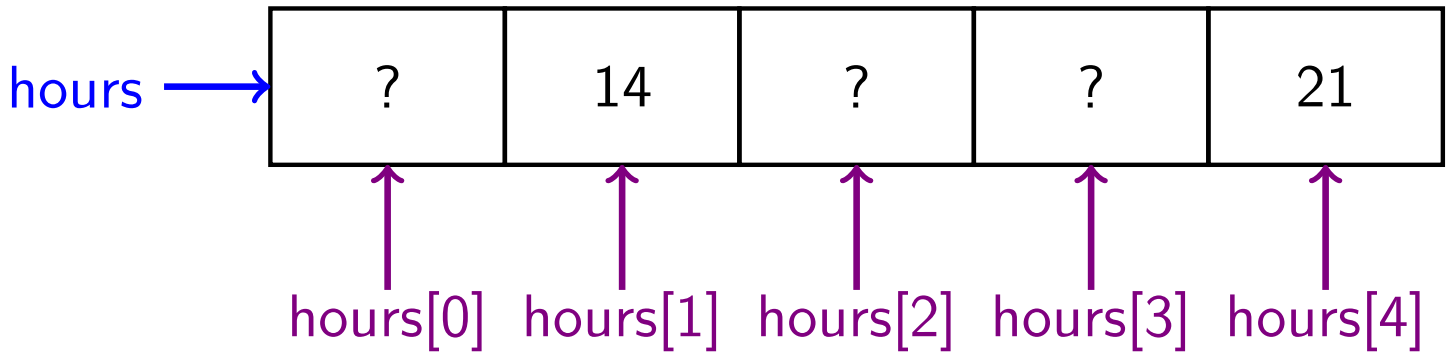
- you cannot use a variable to declare an array's size

```
unsigned score[number_of_scores];
```

- an array's size must be specified by a literal or a constant (or implicit via initialization)
- since a literal will likely be a magic number, use a constant to declare an array's size

Initializing Individual Elements

```
const unsigned ARRAY_SIZE = 5;
unsigned hours[ARRAY_SIZE];
```



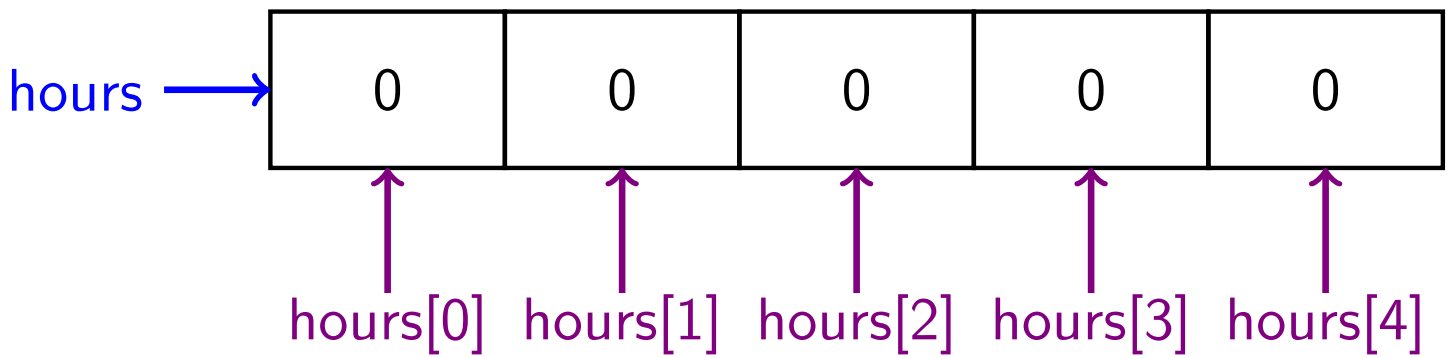
- just like every other variable, array elements are not initialized until the program specifically gives them a value
- they can be given values individually one-by-one:

```
hours[1] = 14;
hours[4] = 21;
```

Initializing Individual Elements

- or in a loop:

```
for (unsigned index = 0; index < ARRAY_SIZE; index++)
{
  hours[index] = 0;
}
```



- note: it is rare to have a program with an array that doesn't use loops — for loops and arrays go together like bears and honey

Initialize the Array

- the phrase initialize a variable normally means at the time of declaration
- an array can be initialized at declaration

Initialize the Array

- the phrase initialize a variable normally means at the time of declaration
- an array can be initialized at declaration

- ALERT! Gaddis uses the old syntax for array initialization
- we will use the correct modern syntax

Initialize the Array

- the phrase initialize a variable normally means at the time of declaration
- an array can be initialized at declaration

- ALERT! Gaddis uses the old syntax for array initialization
- we will use the correct modern syntax

```
const unsigned NUMBER_OF_MONTHS = 12;
unsigned days[NUMBER_OF_MONTHS] {31, 28, 31, 30,
                                 31, 30, 31, 31,
                                 30, 31, 30, 31};
```

- note there is no assignment operator before the curly brace
- this is the new syntax, different from Gaddis
- note there is a semicolon after the closing curly brace

see program_7_5.cpp

Implicit Array Sizing

- if you provide an initialization list, you do not need to specify the size of the array
  `double ratings[] {1.0, 1.5, 3.3, 2.6, 0.9};`
- the compiler can count the size of the initialization list and know that the full declaration is
  `double ratings[5] {1.0, 1.5, 3.3, 2.6, 0.9};`

Bounds Checking

- it is illegal to reference an array element that does not exist

```
int foo[10];
foo[10] = 0; // illegal!  largest index is 9!
```

Bounds Checking

- it is illegal to reference an array element that does not exist

```
int foo[10];
foo[10] = 0; // illegal!  largest index is 9!
```

- this will eventually cause you grief
- but sometimes you won't notice it right away

Bounds Checking

- it is illegal to reference an array element that does not exist

```
int foo[10];
foo[10] = 0; // illegal!  largest index is 9!
```

- this will eventually cause you grief
- but sometimes you won't notice it right away

- on ice, program_7_9.cpp will not crash at all with TOO_MANY set to 5
- it will crash (but not immediately) with TOO_MANY set to 20

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 (auto value : 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

The Range-Based for Loop

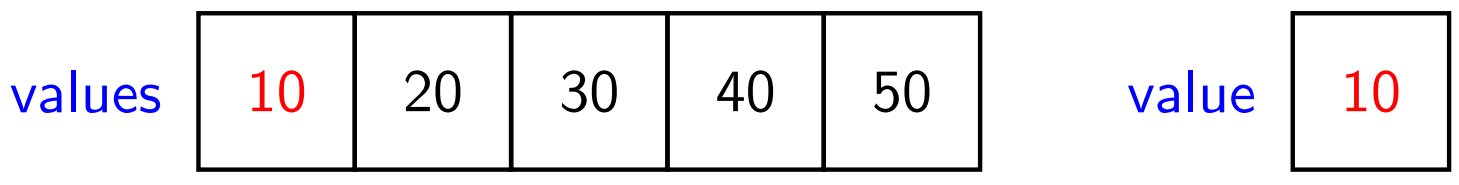
```
int values[] {10, 20, 30, 40, 50};
for (auto value : values)
{
  ... do something with value
}
```

- the foreach loop is almost always used with the auto keyword
- you could put a datatype there:
  `for (int value : values)`
- but since the compiler is already figuring out the index and the size, you might as well let it figure out the correct data type also

foreach Under the Hood

```
int values[] {10, 20, 30, 40, 50};
for (auto value : values)
{
   ... do something with value
}
```

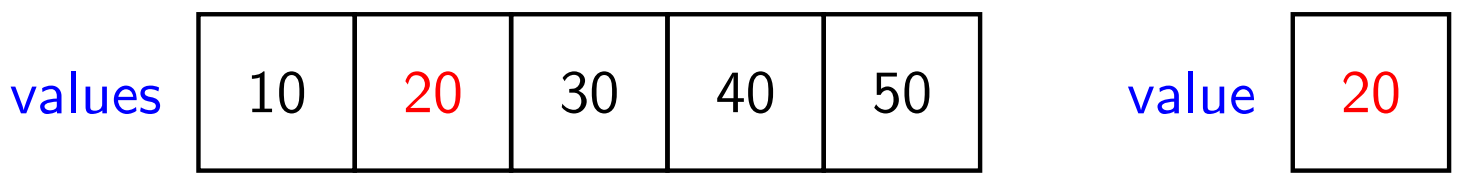
- value is a separate variable
- each time through the loop, it gets a copy of the next element of the array



foreach Under the Hood

```
int values[] {10, 20, 30, 40, 50};
for (auto value : values)
{
   ... do something with value
}
```

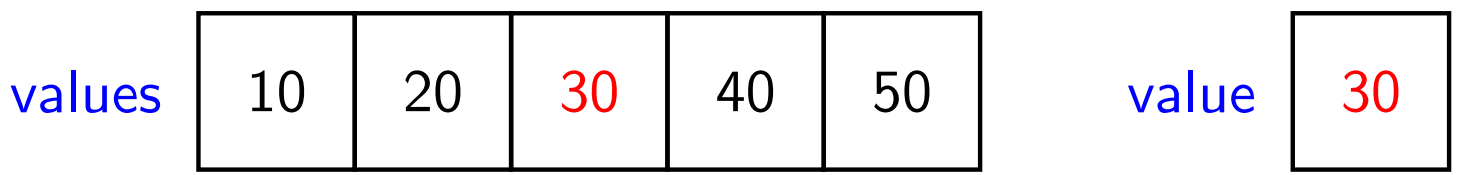
- value is a separate variable
- each time through the loop, it gets a copy of the next element of the array



foreach Under the Hood

```
int values[] {10, 20, 30, 40, 50};
for (auto value : values)
{
   ... do something with value
}
```

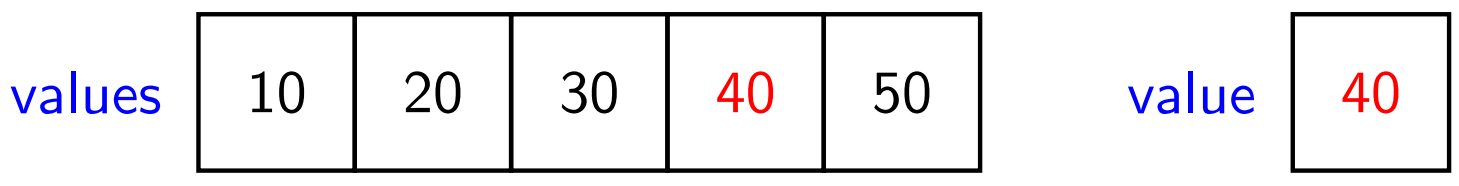
- value is a separate variable
- each time through the loop, it gets a copy of the next element of the array



foreach Under the Hood

```
int values[] {10, 20, 30, 40, 50};
for (auto value : values)
{
   ... do something with value
}
```

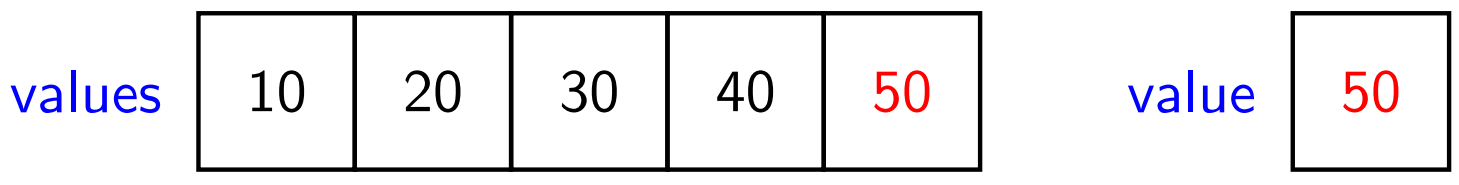
- value is a separate variable
- each time through the loop, it gets a copy of the next element of the array



foreach Under the Hood

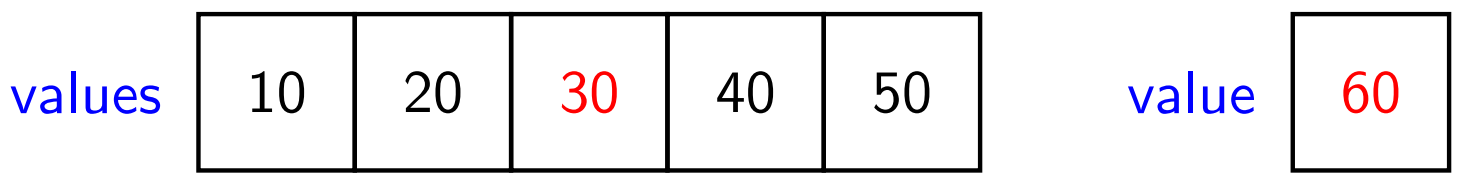
```
int values[] {10, 20, 30, 40, 50};
for (auto value : values)
{
   ... do something with value
}
```

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



foreach Under the Hood

- value 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:

```
int values[] {10, 20, 30, 40, 50};
for (auto value : values)
{
  value += 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

```
int values[] {10, 20, 30, 40, 50};
for (auto& value : values)
{
  value += 10;
}
```

- now value is not a copy of the array element
- it is an alias of the array element
- a change made to value 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 need to have the indices of the array elements

```
for (auto value : values)
{
  cout << value << 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 you 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 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?
```

Whole-Array Comparison

- I 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?

```
bool same = true;
unsigned index = 0;
while (same && index < SIZE)
{
  same = array1[index] == array2[index];
  index++;
}
// here, same is either true or false
```

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

- sum the contents

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

Common Array Algorithms

- compute the average

```
double total = 0.0;
for (auto value : values)
{
  total += value;
}
double average = total / NUMBER_OF_VALUES;
```

Common Array Algorithms

- find the largest value

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

Common Array Algorithms

- find the position of the largest value
- remember, if you need the index, can't use foreach

```
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;
  }
}
```

How Big?

- it is extremely common to not know in advance how many values you will need to store in an array
- how many students will be in the file?
- how many scores will each student have?
- if you don't know how many scores there will be, how big should you make the array?

How Big?

- it is extremely common to not know in advance how many values you will need to store in an array
- how many students will be in the file?
- how many scores will each student have?
- if you don't know how many scores there will be, how big should you make the array?

- remember, an array's size is constant
- you cannot resize an array

How Big?

- it is extremely common to not know in advance how many values you will need to store in an array
- how many students will be in the file?
- how many scores will each student have?
- if you don't know how many scores there will be, how big should you make the array?

- remember, an array's size is constant
- you cannot resize an array

- the solution is to make the array big enough to hold the largest possible number of values in the situation

Partially Filled Arrays

- a partially filled array is an array with some valid values
- and some unused spaces because the array is bigger than necessary at the moment
- for this to work, you must keep track of how many values are valid
- you do this using a separate count variable
- and to be safe, you must make sure you aren't exceeding the array's size

see program_7_14.cpp

Parallel Arrays

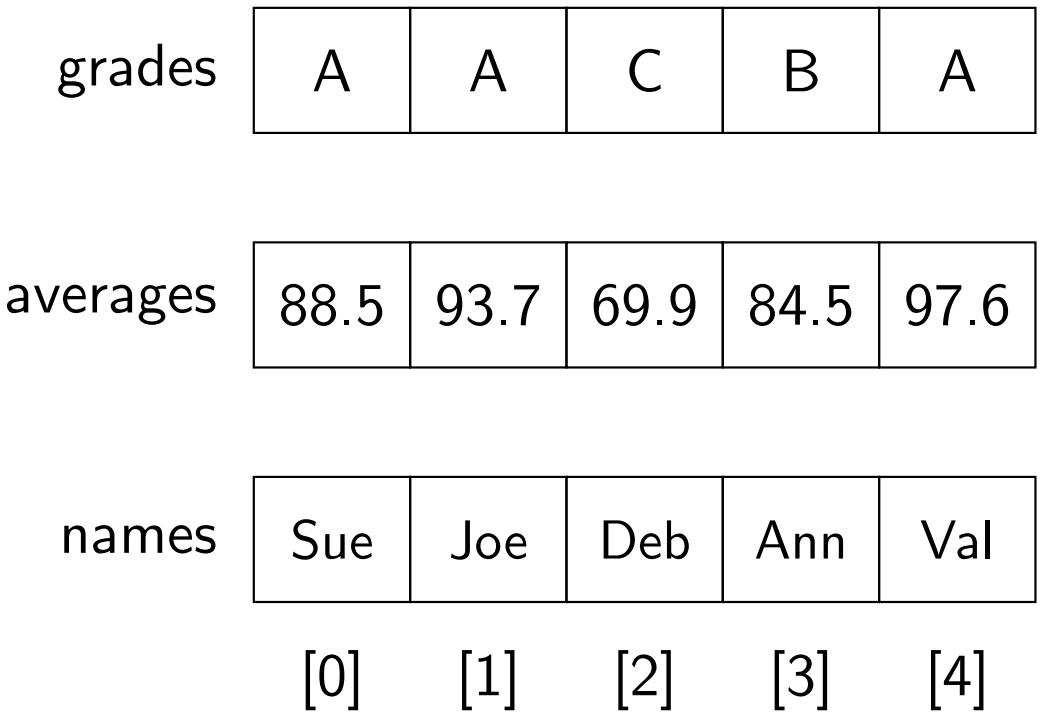
- imagine you are maintaining a gradebook for students in a class
- each student has a
  - name (a string)
  - current overall average (a double)
  - current grade (a char)

Parallel Arrays

- imagine you are maintaining a gradebook for students in a class
- each student has a
  - name (a string)
  - current overall average (a double)
  - current grade (a char)

- we can use parallel arrays to manage this information easily

Parallel Arrays



- a subscript represents a student
- one student's data is spread across three arrays
- at the same location in each array