Introduction to Programming with C++

Arrays and Loops

Objective

- In this chapter, you'll learn:
  - What an array is and how you create an array
  - How to use a for loop
  - How the while loop works
  - What the merits of the do-while loop are
  - What the break and continue statement do in a loop
  - What the continue statement does in a loop
  - How to use nested loops
  - How to create and use an array container
  - How to create and use a vector container

Introduction

- An array enables you to work with several data items of the same type using a single name, the array name.
- The need for this occurs often
- For example, working with a series of temperatures or the ages of a group of people.
- A loop is another fundamental programming facility.
- It provides a mechanism for repeating one or more statements as many times as your application requires.
- For example calculating payroll for all employees of a company would make use of a loop.
- There are several kinds of loop, each with their own particular area of application.

Arrays

- The variables you have created up to now can store only a single data item of the specified type
- An array stores several data items of the same type.
- You can create an array of any type of data
- There can be as many as the available memory will allow.
- An array is a variable that represents a sequence of memory locations
- Each can store an item of data of the same data type
- The data values are called ***elements***.

- Suppose you've written a program to calculate an average temperature.
- You now want to extend the program to calculate how many samples are above the average and how many are below.
- You'll need to retain the original sample data to do this, but storing each data item in a separate variable would be tortuous to code and impractical for anything more than a very few items.
- An array provides you with the means of doing this easily, and many other things besides.

Using an Array

- For example, you could store 366 temperature samples in an array defined as follows:

```
double temperatures[366]; // An array of temperatures
```

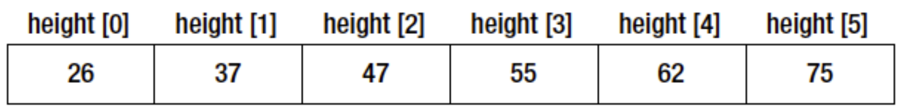
- This defines an array with the name temperatures to store 366 values of type `double`.
- The number of elements specified between the brackets is the ***size of the array***.
- The array elements are not initialized in this statement so they contain junk values.
- You refer to an array element using an integer called an ***index***.

Using an Array

- The index of a particular array element is its offset from the first element.
- The first element has an offset of 0 and therefore an index of 0;
- an index value of 3 refers to the fourth array element — three elements from the first.
- To reference an element, you put its index between square brackets after the array name
- So to set the fourth element of the temperatures array to 99.0, you would write:

```
temperatures[3] = 99.0;
```

Using an Array



- The figure above shows the structure of an array called height that has six elements of type int.
- The array has six elements of type int.
- Each box represents a memory location holding an array element.
- Each element can be referenced using the expression above it.
- You can define an array that has six elements of type int using this statement:

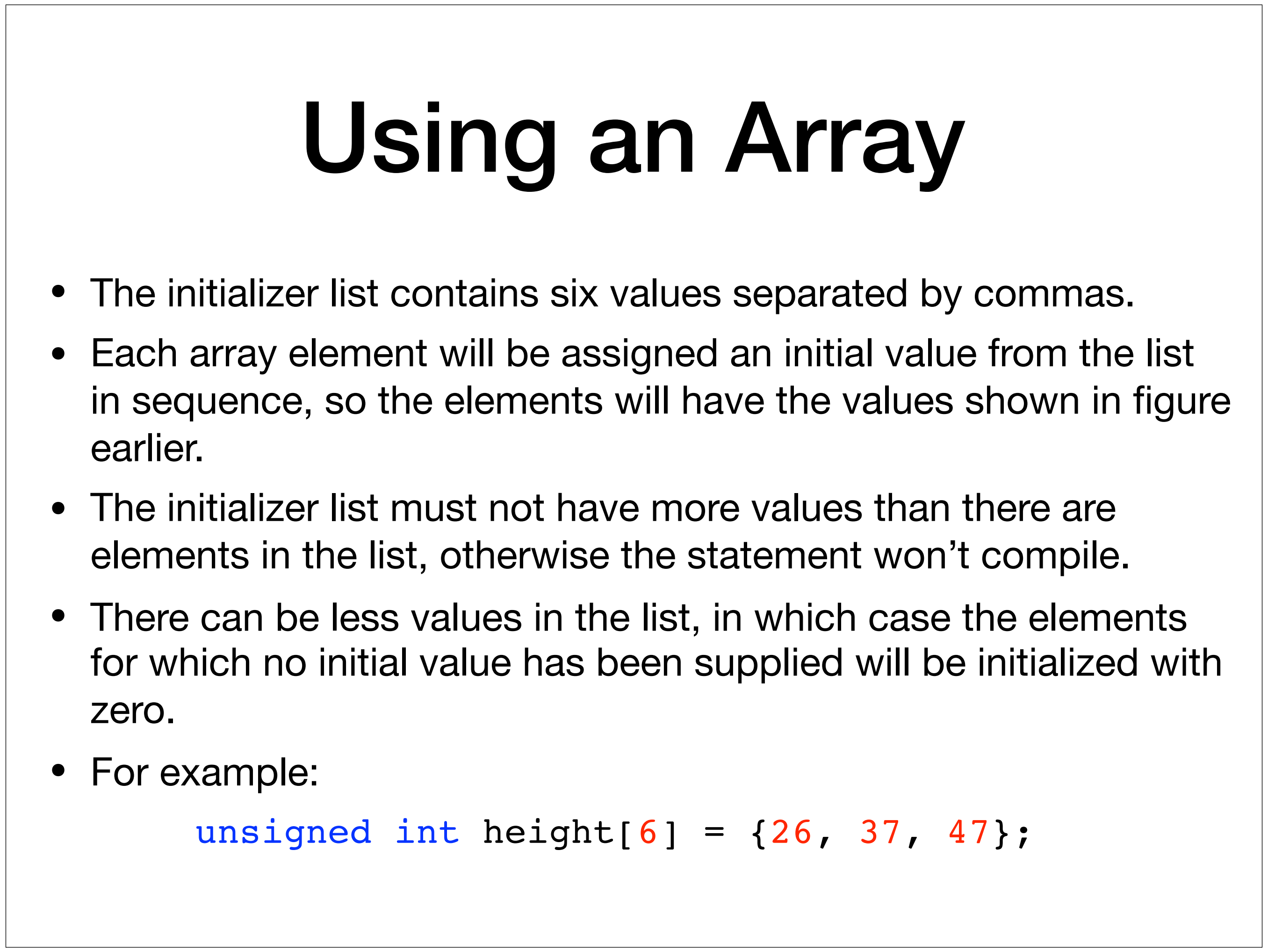
```
unsigned int height[6]; // Define an array of six heights
```

- The compiler will allocate six contiguous storage locations for storing values of type unsigned int as a result of this definition.
- The definition doesn't specify any initial values for the array, so the elements contain junk values.

Using an Array

- The type of the array will determine the amount of memory required for each element.
- The elements of an array are stored in one contiguous block of memory
- Each element in the height array contains a different value.
- These might be the heights of the members of a family, recorded to the nearest inch.
- As there are six elements, the index values run from 0 for the first element through to 5 for the last element.
- You could define the array with these initial values like this:

```
unsigned int height[6] {26, 37, 47, 55, 62, 75};
```



Using an Array

- The initializer list contains six values separated by commas.
- Each array element will be assigned an initial value from the list in sequence, so the elements will have the values shown in figure earlier.
- The initializer list must not have more values than there are elements in the list, otherwise the statement won't compile.
- There can be less values in the list, in which case the elements for which no initial value has been supplied will be initialized with zero.
- For example:

```
unsigned int height[6] = {26, 37, 47};
```

- The first three elements will have the values that appear in the list.
- The last three will be zero.

Using an Array

- To initialize all the elements with zero, you can just use an empty initializer list:

```
unsigned int height[6] {};
```

- Array elements participate in arithmetic expressions like other variables.
- You could sum the first three elements of height like this:

```
unsigned int sum {};
sum = height[0] + height[1] + height[2];
```

Using an Array

- You use references to individual array elements like ordinary integer variables in an expression.
- An array element can be on the left of an assignment to set a new value so you can copy the value of one element to another in an assignment
- For example:

```
height[3] = height[2]; // Copy 3rd element value to 4th element
```

- However, you can't copy all the element values from one array to the elements of another in an assignment.
- You can only operate on individual elements.
- To copy the values of one array to another, you must copy the values one at a time.
- What you need is a loop.

Understanding Loops

- A loop is a mechanism that enables you to execute a statement or block of statements repeatedly until/while a particular condition is met.
- The statements inside a loop are sometimes called iteration statements.
- A single execution of the statement or statement block that is within the loop is an iteration.
- Two essential elements make up a loop:
  - The statement or block of statements that forms the body of the loop that is to be executed repeatedly
  - A loop condition of some kind that determines when to stop repeating the loop.

Understanding Loops

- A loop condition can take different forms to provide different ways of controlling the loop.
- For example, a loop condition can:
  - Execute a loop a given number of times.
  - Execute a loop until a given value exceeds another value.
  - Execute the loop until a particular character is entered from the keyboard.
  - Execute a loop for each element in a collection of elements.
- You choose the loop condition to suit the circumstances.

Understanding Loops

- You have the following varieties of loops:
    - The for loop primarily provides for executing the loop a prescribed number of times but there is considerable flexibility beyond that.
    - The range-based for loop executes one iteration for each element in a collection of elements.
    - The while loop continues executing as long as a specified condition is true.
    - The condition is checked at the beginning of an iteration so if the condition starts out as false, no loop iterations are executed.

Understanding Loops

- The do-while loop continues to execute as long as a given condition is true.
- This differs from the while loop in that the do-while loop checks the condition at the end of an iteration.
- This implies that at least one loop iteration always executes.
- I’ll start by explaining how the for loop works.

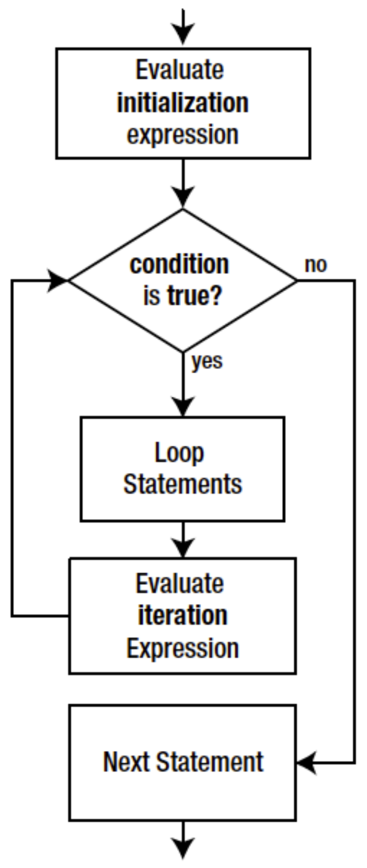
The for Loop

- The for loop executes a statement or block of statements a predetermined number of times
- But you can use it in other ways too.
- You specify how a for loop operates using three expressions separated by semicolons between parentheses following the `for` keyword.

```
for(initialization ; condition ; iteration)
{

  // Loop statements

}
// Next statement
```


Evaluate initialization expression
condition is true?
no
yes
Loop Statements
Evaluate iteration Expression
Next Statement

The for Loop

- You can omit any or all of the expressions controlling a for loop but you must always include the semicolons.
- We'll explain later in this chapter why and when you might omit one or other of the control expressions.
- The initialization expression is evaluated only once, at the beginning of the loop.
- The loop condition is checkednext, and if it is true, the loop statement or statement block executes.
- If the condition is false, the loop ends and execution continues with the statement after the loop.
- After each execution of the loop statement or block, the iteration expression is evaluated and the condition is checked to decide if the loop should continue.

The for Loop

- In the most typical usage of the for loop
- the first expression initializes a counter
- the second expression checks whether the counter has reached a given limit
- the third expression increments the counter.
- For example, you could copy the elements from one array to another like this:

```
double rainfall[12] {1.1, 2.8, 3.4, 3.7, 2.1, 2.3,
                     1.8, 0, 0.3, 0.9, 0.7, 0.5};
double temp[12] {};
for(size_t i {} ; i<12 ; ++i){
      temp[i] = rainfall[i];
}
```

- The first expression defines i as type `size_t` with an initial value of 0.

The for Loop

- You'll recall that the sizeof operator returns a value of `size_t`,
- which is an unsigned integer type that is used generally for sizes of things as well as counts.
- i will be used to index the arrays so using `size_t` makes sense.
- Not only is it legal to define variables within a for loop initialization expression, it is very common.
- This has some significant implications.
- A loop defines a scope.
- The loop statement or block, including any expressions that control the loop fall within the scope of a loop.

The for Loop

- Any automatic variables declared within the scope of a loop do not exist outside it.
- Because i is defined in the first expression, it is local to the loop so when the loop ends, i will no longer exist.
- The second expression, the loop condition, is true as long as i is less than 12, so the loop continues while i is less than 12.
- When i reaches 12, the expression will be false so the loop ends.
- The third expression increments i at the end of each loop iteration so the loop block that copies the i^{th} element from rainfall to temp will execute with values of i from 0 to 11.

The for Loop

- When you need to be able to access the loop control variable after the loop ends, you just define it before the loop, like this:

```
size_t i {};
for(i = 0; i<12 ; ++i){
     temp[i] = rainfall[i];
}
// I still exists here
```

- The first expression defines i as type `size_t` with an initial value of 0.

Avoiding Magic Numbers

- One problem with the preceding code fragment is that it involves the "magic number" 12 for the array sizes.
- It's easy to make a mistake when entering the rainfall array size of 12 in the definition of the temp array and in the for loop.
- It would be better to define a const variable for the array size and use that instead of the explicit value:

```
const size_t size {12};
double rainfall[size] {1.1, 2.8, 3.4, 3.7, 2.1, 
2.3,
                    1.8, 0, 0.3, 0.9, 0.7, 0.5};
double temp[size] {};
for(size_t i = 0; i < size; ++i){
      temp[i] = rainfall[i];
}
```

See Ex5_01.cpp

Array Index

- Array index values are not checked to verify that they are valid.
- It’s up to you to make sure that you don’t reference elements outside the bounds of the array.
- If you store data using an index value that’s outside the valid range for an array, you’ll overwrite something in memory or cause a storage protection violation.
- Either way, your program will almost certainly come to a sticky end.

Defining the Array Size with the Initializer List

- You can omit the size of the array when you supply one or more initial values in its definition.
- The number of elements will be the number of initial values. For example:

```
int values[] {2, 3, 4};
```

- This defines an array with three elements of type int that will have the initial values 2, 3, and 4.
- It is equivalent to writing this:

```
int values[3] {2, 3, 4};
```

- The advantage of omitting the size is that you can't get the array size wrong the compiler determines it for you.
- You can't have an array with no elements so the initializer list must always contain at least one initial value if you omit the array size.
- An empty initializer list will result in a compilation error if you don't specify the array size.

Determining the Size of an Array

- You saw earlier how you could avoid magic numbers for the number of elements in an array by defining a constant initialized with the array size.
- You also don't want to be specifying a magic number for the array size when you let the
- compiler decide the number of elements from the initializer list.
- You need a foolproof way of determining the size when necessary.
- The sizeof operator returns the number of bytes that a variable occupies and this works with an entire array
- as well as with a single array element.
- Thus the sizeof operator provides a way to determine the number of elements in an array
- You just divide the size of the array by the size of the first element.
- Suppose you've defined this array:

```
int values[] {2, 3, 5, 7, 11, 13, 17, 19};
```

See Ex5_02.cpp, Ex5_03.cpp, and Ex5_04.cpp

The Comma Operator

- Although the comma looks as if it’s just a humble separator, it is actually a binary operator.
- It combines two expressions into a single expression, where the value of the operation is the value of its right operand.
- This means that anywhere you can put an expression, you can also put a series of expressions separated by commas.
- For example, consider the following statements:

```
int i {1};
int value1 {1};
int value2 {1};
int value3 {1};
std::cout << (value1 += ++i, value2 += ++i,
              value3 += ++i) << std::endl;
```

The Comma Operator

- The first four statements define four variables with an initial value 1.
- The last statement outputs the result of three assignment expressions that are separated by the comma operator.
- The comma operator is left associative and has the lowest precedence of all the operators so the expression evaluates like this:

```
(((value1 += ++i), (value2 += ++i)), (value3 += ++i));
```

- The effect will be that value1 will be incremented by 2 to produce 3, value2 will be increments by 3 to produce 4, and value3 will be incremented by 4 to produce 5.
- The value of the composite expression is the value of the rightmost expression in the series, so the value that is output is 5.

The Ranged-based for Loop

- The range-based for loop iterates over all the values in a range of values.
- This raises the immediate question: what is a range?
- An array is a range of elements and a string is a range of characters.
- The containers provided by the Standard Library for managing are all ranges.
- We’ll introduce two Standard Library containers later.
- The general form of the range-based for loop is:

```
for(range_declaration : range_expression}
     loop statement or block;
```

The Ranged-based for Loop

- The range_declaration identifies a variable that will be assigned each of the values in the range in turn
- The range_expression identifies the range that is the source of the data.
- Consider these statements:

```
int values [] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
int total {};
for(int x : values)
  total += x;
```

- The variable x will be assigned a value from the values array on each iteration.
- It will be assigned values 2, 3, 5, and so on in succession.
- Thus the loop will accumulate the sum of all the elements in the values array in total.
- The variable x is local to the loop and does not exist outside it.

The Ranged-based for Loop

- The compiler knows the type of the elements in the values array
- So you could allow the compiler to determine the type for x by writing the loop as:

```
for(auto x : values)
  total += x;
```

- Using the `auto` keyword causes the compiler to deduce the correct type for x.
- The `auto` keyword is used very often with the range-based for loop.
- This is a very nice way of iterating over all the elements in an array or other kind of range.
- You don’t need to be aware of the number of elements.
- The loop mechanism takes care of that.

The Ranged-based for Loop

- Note that the values from the range are assigned to the range variable, x.
- This means that you cannot modify the elements of values by modifying the value of x.
- For example, this doesn't change the elements in the values array:

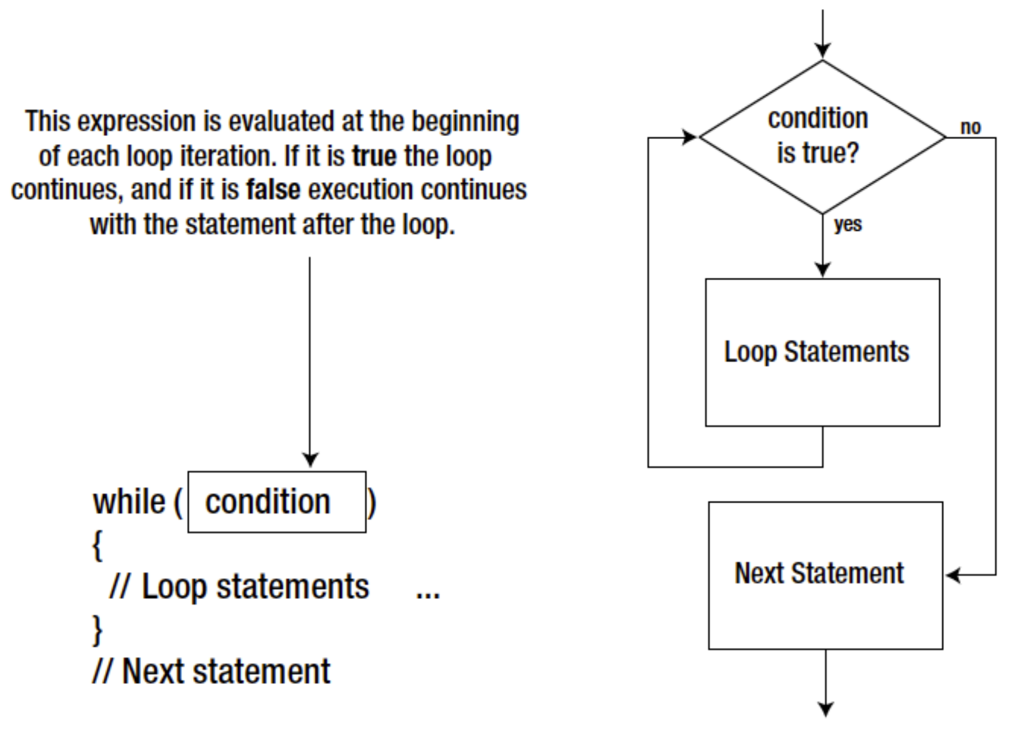
```
for(auto x : values)
  x += 2;
```

- This just adds 2 to the local variable, x, not to the array element.
- The value stored in x is overwritten by the value of the next element from values on the next iteration.

The while Loop

- The while loop uses a logical expression to control execution of the loop body.
- You can use any expression to control the loop, as long as it evaluates to a value of type bool, or can be implicitly converted to type bool.
- If the loop condition expression evaluates to a numerical value for example, the loop continues as long as the value is non-zero.
- A zero value ends the loop.

The while Loop


This expression is evaluated at the beginning of each loop iteration. If it is **true** the loop continues, and if it is **false** execution continues with the statement after the loop.

```
while ( condition )
{
  // Loop statements    ...
}
// Next statement
```

See Ex5_05.cpp

Allocating an Array at Runtime

- The C++14 standard does not permit an array dimension to be specified at runtime
- the array dimension must be a constant expression that can be evaluated by the compiler.
- However, some current C++ compilers do allow array dimensions at runtime because the current C standard, C99, permits this and a C++ compiler will typically compile C code too.
- The view at present is that this feature may be added to C++ in a future standard specification.
- Determining the size of an array is a very useful feature if your compiler supports it see Ex5_06.cpp
- Keep in mind though that this is not strictly in conformance with the C++ language standard.

The do-while Loop

- The do-while loop is similar to the while loop in that the loop continues for as long as the specified loop condition remains true.
- However, the difference is that the loop condition is checked at the end of the do-while loop, rather
- than at the beginning
- So the loop statement is always executed at least once.
- The logic and general form of the do-while loop are shown in the next slide.
- Note that the semicolon that comes after the condition between the parentheses is absolutely necessary.
- If you leave it out, the program won’t compile.

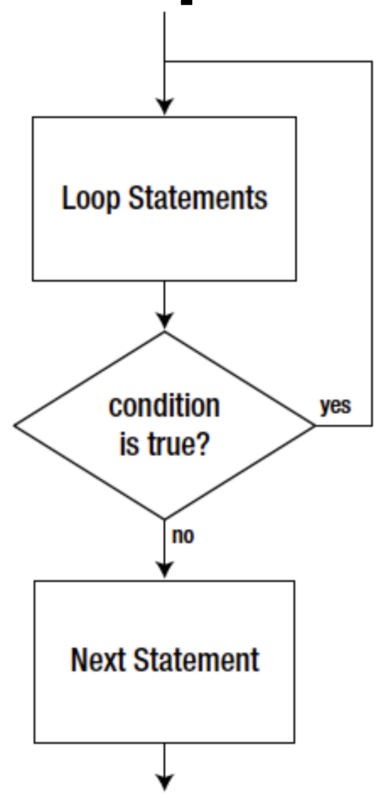
The do-while Loop

```
do
{
  // Loop statements    ...

}while ( condition );

// Next Statement
```

This expression is evaluated at the end of each loop iteration. If it is **true** the loop continues, and if it is **false** execution continues with the statement after the loop. The loop statements are always executed at least once.



See Ex5_07.cpp

Nested Loops

- You can place a loop inside another loop.
- In fact, you can nest loops within loops to whatever depth you require to solve your problem.
- Furthermore, nested loops can be of any kind
- You can nest a for loop inside a while loop inside a do-while loop inside a range-based for loop, if you have the need.
- They can be mixed in any way that you want.
- Nested loops are often applied in the context of arrays but they have many other uses.

See Ex5_08.cpp

Skipping Loop Iterations

- Situations arise where you want to skip one loop iteration and press on with the next.
- The continue statement does this:

```
continue; // Go to the next iteration
```

- When this statement executes within a loop, execution transfers immediately to the end of the current iteration.
- As long as the loop control expression allows it, execution continues with the next iteration.

See Ex5_09.cpp

Breaking Out of a Loop

- Sometimes, you need to end a loop prematurely;
- something might arise within the loop statement that indicates there is no point in continuing.
- In this case, you can use the `break` statement.
- Its effect in a loop is much the same as it is in a switch statement;
- executing a `break` statement within a loop ends the loop immediately and execution continues with the statement following the loop.
- The break statement is used most frequently with an indefinite loop, so let's look next at what one of those looks like.

Indefinite Loops

- An indefinite loop can potentially run forever.
- Omitting the second control expression in a for loop results in a loop that potentially executes an unlimited number of iterations.
- There has to be some way to end the loop within the loop block itself; otherwise the loop repeats indefinitely.
- Indefinite loops have many practical uses:
  - programs that monitor some kind of alarm indicator for instance
  - that collect data from sensors in an industrial plant.
- An indefinite loop can be useful when you don’t know in advance how many loop iterations will be required
- Such as when you are reading a variable quantity of input data.

Indefinite Loops

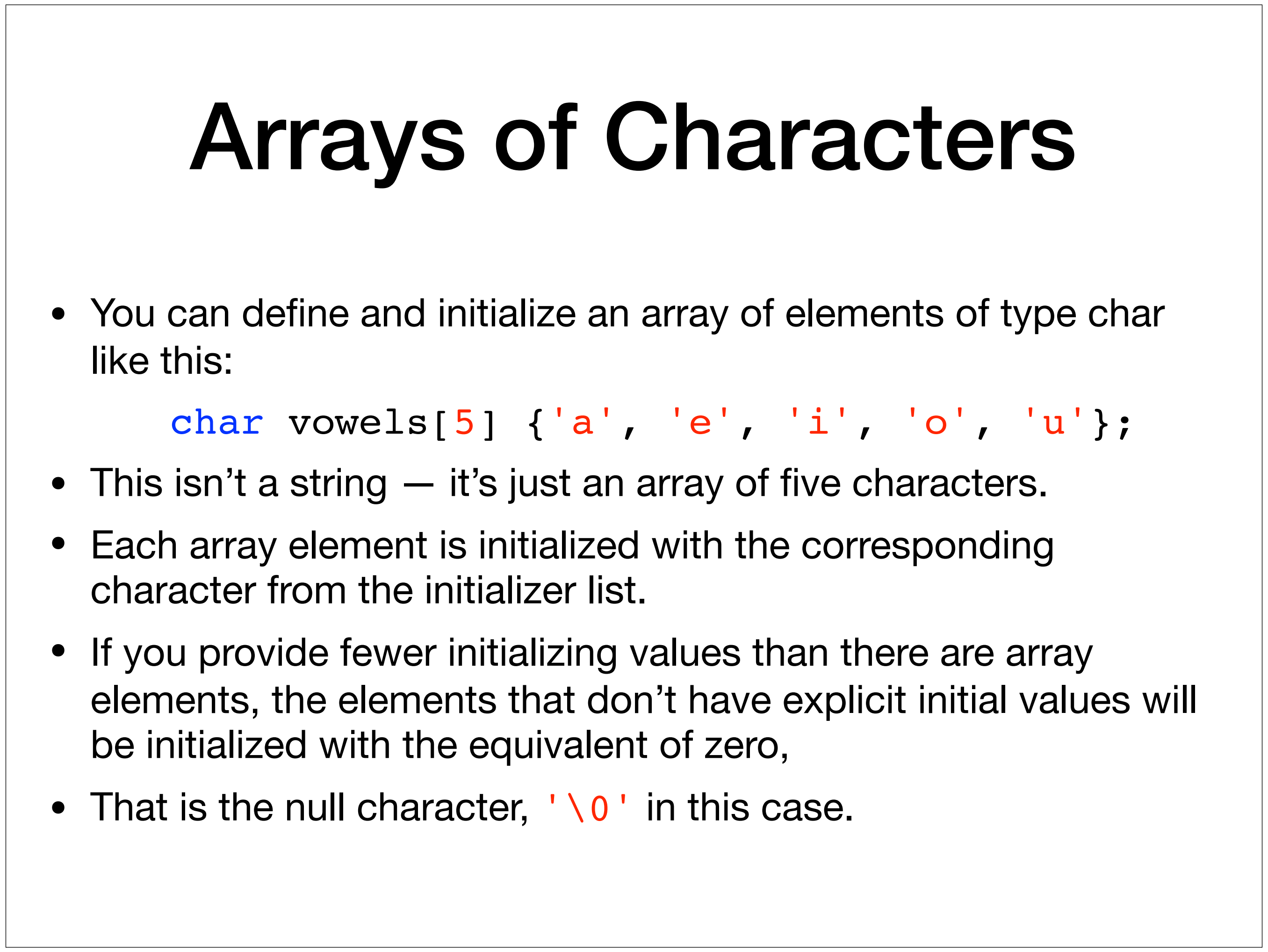
- In these circumstances, you code the exit from the loop within the loop block, not within the loop control expression.
- In the most common form of the indefinite for loop, all the control expressions are omitted, as shown here:

```
for( ; ; ) {
   // Statements that do something...
   // and include some way of ending the loop
}
```

- You still need the semicolons (;), even though no loop control expressions exist.
- The only way this loop can end is if some code within the loop terminates it.

Arrays of Characters

- An array of elements of type char can have a dual personality.
- It can simply be an array of characters, in which each element stores one character, or it can represent a string.
- Characters in a string are stored in successive array elements, followed by a special string termination character called the null character that you write as `'\0'`
- this marks the end of the string.
- A character string that is terminated by `'\0'` is a C-style string
- This contrasts with the string type from the Standard Library that we'll see later in this course



Arrays of Characters

- You can define and initialize an array of elements of type char like this:

```
char vowels[5] {'a', 'e', 'i', 'o', 'u'};
```

- This isn't a string — it's just an array of five characters.
- Each array element is initialized with the corresponding character from the initializer list.
- If you provide fewer initializing values than there are array elements, the elements that don't have explicit initial values will be initialized with the equivalent of zero,
- That is the null character, `'\0'` in this case.

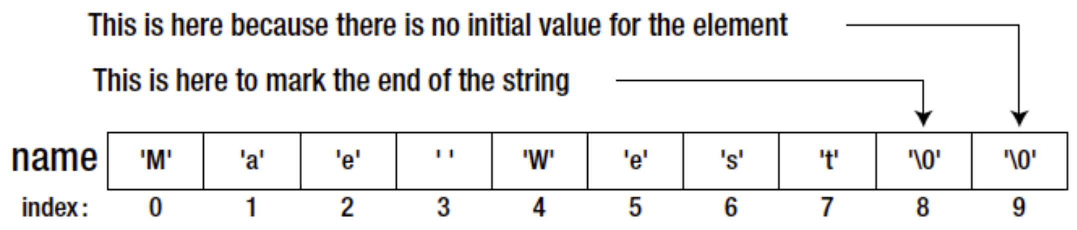
- This means that if there are insufficient initial values, the array will effectively contains string.
- For example:
- char vowels[6] {'a', 'e', 'i', 'o', 'u'};
- The last element will be initialized with '\0'.
- The presence of the null character means that this can be treated as a C-style string. Of course, you can still regard it as an array of characters.

Arrays of Characters

- You can also declare an array of type char and initialize it with a string literal
- For example:

```
char name[10] {”Mae West”};
```

- This creates a C-style string.
- Because you’re initializing the array with a string literal, the null character will be
- stored in the element following the last string character, so the contents of the array will be as shown below



This is here because there is no initial value for the element
This is here to mark the end of the string
name
'M'
'a'
'e'
' '
'W'
'e'
's'
't'
'\0'
'\0'
index:
0
1
2
3
4
5
6
7
8
9

- Of course, you can leave the compiler to set the size of the array when you initialize it with a string:

```
char name[] {"Mae West"};
```

- This time, the array will have nine elements: eight to store the characters in the string, plus an extra element to store the string termination character.

Arrays of Characters

- You can output a string stored in an array just by using the array name.
- The string in the name array, for example, could be written to `cout` with this statement:

```
std::cout << name << std::endl;
```

- This will display the entire string of characters, up to the `'\0'`.
- There must be a `'\0'` at the end.
- If there isn't, you'll continue to output characters from successive memory locations until a string termination character turns up or an illegal memory reference occurs.
- **Note that this only works for char arrays and you can't output the contents of an array of a numeric type by just using the array name.**

See Ex5_11.cpp

Multidimensional Arrays

- All the arrays so far have required a single index value to select an element.
- Such an array is called a one-dimensional array, because varying one index can reference all the elements.
- You can also define arrays that require two or more index values to access an element.
- These are referred to generically as multidimensional arrays.
- An array that requires two index values to reference an element is called a two-dimensional array.
- An array needing three index values is a three-dimensional array, and so on for as many dimensions as you think you can handle.

Multidimensional Arrays

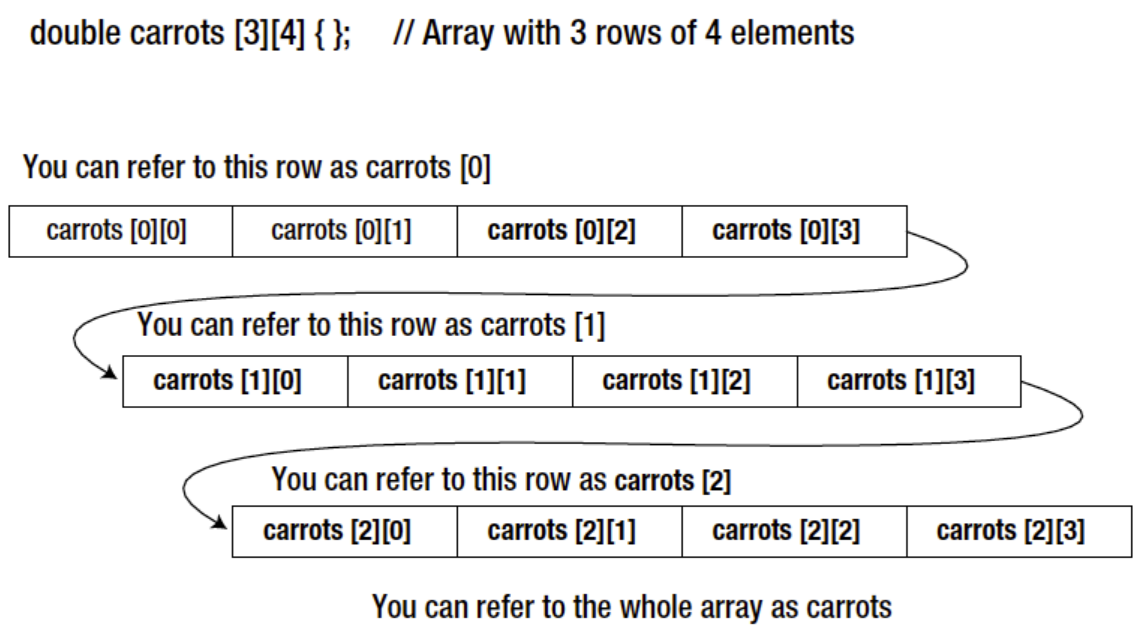
- Suppose that you want to record the weights of the carrots you grow in your small vegetable garden.
- To store the weight of each carrot, which you planted in three rows of four, you could define a two-dimensional array:

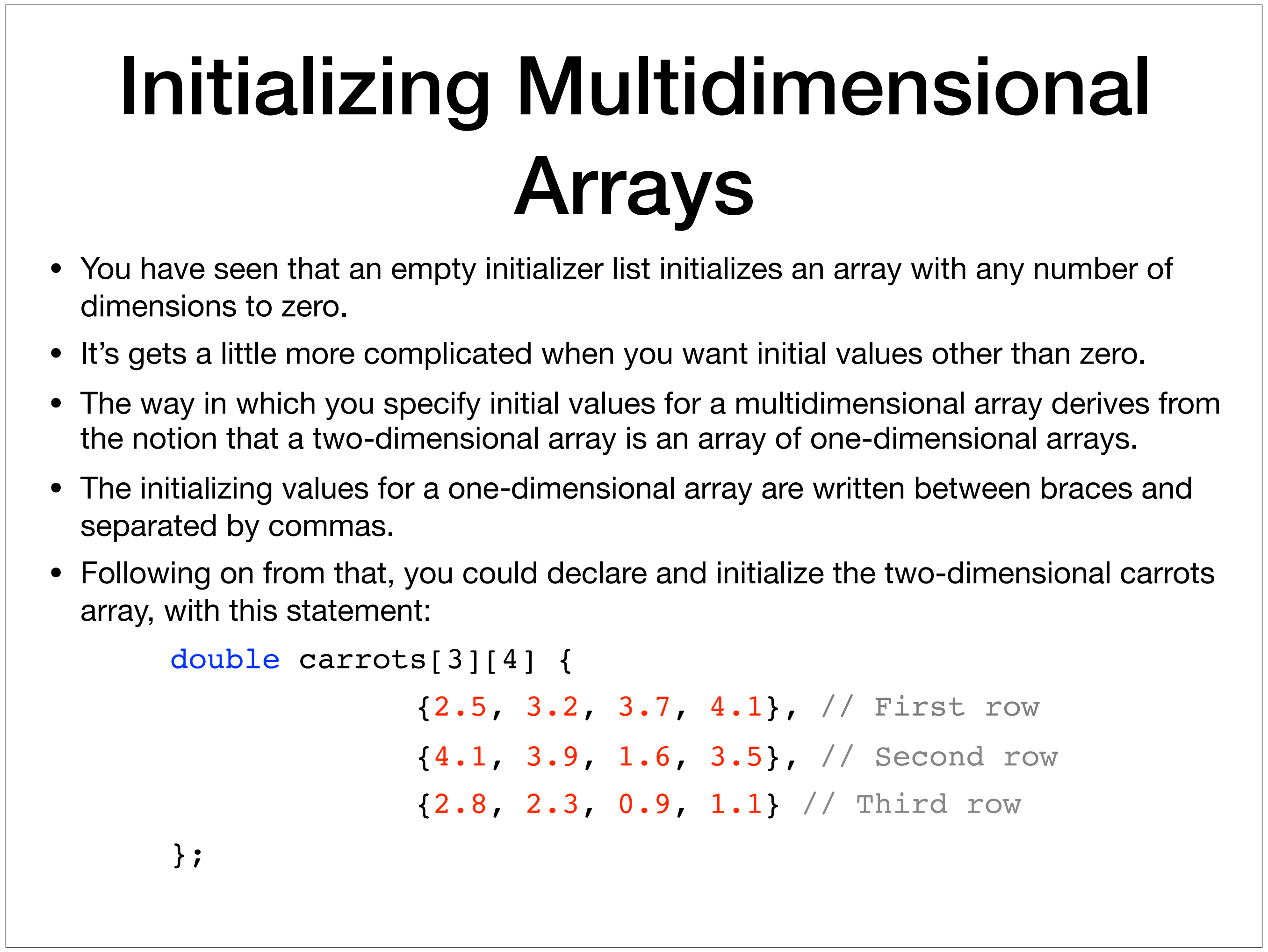
```
double carrots[3][4] {};
```

- This defines an array with 3 rows of 4 elements and initializes all elements to zero.
- To reference a particular element of the carrots array, you need two index values.
- The first index specifies the row, from 0 to 2, and the second index specifies a particular carrot in that row, from 0 to 3.
- To store the weight of the third carrot in the second row, you could write:

```
carrots[1][2] = 1.5;
```

Multidimensional Arrays


double carrots [3][4] { }; // Array with 3 rows of 4 elements
You can refer to this row as carrots [0]
carrots [0][0]
carrots [0][1]
carrots [0][2]
carrots [0][3]
You can refer to this row as carrots [1]
carrots [1][0]
carrots [1][1]
carrots [1][2]
carrots [1][3]
You can refer to this row as carrots [2]
carrots [2][0]
carrots [2][1]
carrots [2][2]
carrots [2][3]
You can refer to the whole array as carrots



Initializing Multidimensional Arrays

- You have seen that an empty initializer list initializes an array with any number of dimensions to zero.
- It's gets a little more complicated when you want initial values other than zero.
- The way in which you specify initial values for a multidimensional array derives from the notion that a two-dimensional array is an array of one-dimensional arrays.
- The initializing values for a one-dimensional array are written between braces and separated by commas.
- Following on from that, you could declare and initialize the two-dimensional carrots array, with this statement:

```
double carrots[3][4] {
             {2.5, 3.2, 3.7, 4.1}, // First row
             {4.1, 3.9, 1.6, 3.5}, // Second row
             {2.8, 2.3, 0.9, 1.1} // Third row
};
```

- Each row is a one-dimensional array, so the initializing values for each row are contained within their own set of braces.
- These three initializer lists are themselves contained within a set of braces, because the two-dimensional array is a one-dimensional array of one-dimensional arrays.
- You can extend this principle to any number of dimensions — each extra dimension requires another level of nested braces enclosing the initial values.

Setting Dimensions by Default

- You can let the compiler determine the size of the first (leftmost) dimension of an array with any number of
- dimensions from the set of initializing values.
- The compiler can only ever determine one of the dimensions in a multidimensional array, and it has to be the first.

Multidimensional Character Arrays

- You can define arrays of two or more dimensions to hold any type of data.
- A two- dimensional array of type char is interesting, because it can be an array of C-style strings.

```
char stars[][80] {
                    "Robert Redford",
                    "Hopalong Cassidy",
                    "Lassie",
                    "Slim Pickens",
                    "Boris Karloff",
                    "Oliver Hardy"
                  };
```

See Ex5_12.cpp

Summary

- The essential points you have learned in this chapter are:
    - An array stores a fixed number of values of a given type.
    - You access elements in a one-dimensional array using an index value between square brackets.
    - Index values start at 0 so an index is the offset from the first element in a one dimensional array
    - An array can have more than one dimension.
    - Each dimension requires a separate index value to reference an element.
    - Accessing elements in an array with two or more dimensions requires an index between square brackets for each array dimension.

Summary

- A loop is a mechanism for repeating a block of statements.
- There are four kinds of loop that you can use: the while loop, the do-while loop, the for loop, and the range-based for loop.
- The while loop repeats for as long as a specified condition is true.
- The do-while loop always performs at least one iteration, and continues for as long as a specified condition is true.
- The for loop is typically used to repeat a given number of times and has three control expressions.
- The first is an initialization expression, executed once at the beginning of the loop.
- The second is a loop condition, executed before each iteration, which must evaluate to true for the loop to continue.

Summary

- The third is executed at the end of each iteration and is usually used to increment a loop counter.
- The range-based for loop iterates over all elements within a range.
- An array is a range of elements and a string is a range of characters.
- The array and vector containers define a range so you can use the range-based for loop to iterate over the elements they contain.
- Any kind of loop may be nested within any other kind of loop to any depth.
- Executing a continue statement within a loop skips the remainder of the current iteration and
- goes straight to the next iteration, as long as the loop control condition allows it.

Summary

- Executing a break statement within a loop causes an immediate exit from the loop.
- A loop defines a scope so that variables declared within a loop are not accessible outside the loop.
- In particular, variables declared in the initialization expression of a for loop are not accessible outside the loop.

Reading Assignment

1. Alternatives to Using an Array Page 140 - 147

Exercise

1. Write a program that outputs the squares of the odd integers from 1 up to a limit that is entered by the user.
2. Write a program that uses a while loop to accumulate the sum of an arbitrary number of integers that are entered by the user. The program should output the total of all the values and the overall average as a floating-point value.
3. Create a program that uses a do-while loop to count the number of non whitespace characters entered on a line. The count should end when the first # character is found.