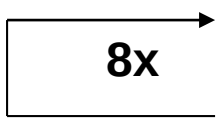
COMSC-110 Chapter 11

Checking it twice: Arrays

Valerie Colber, MBA, PMP, SCPM

Partial algorithm to average 8 scores



open file “8Scores.txt” for input
create scoreTotal and initialize to 0
read a score from the file
add the score to scoreTotal
close the input file
calculate the average score
print the average score with a label

Input file 8scores.txt

66
98
87
71
56
82
84
90

avgFile.cpp

```
…
#include <fstream>
#include <iostream>
using namespace std;
…
int main()
{
  //data
  ifstream fin;  //represents input file
  int scoreTotal = 0;  //sum of all scores read
  int i; // loop counter
  int aScore;  //each score read
  double average=0;  //average score
…
  // read the scores and build the sum
  fin.open("8Scores.txt");
  if (!fin.good()) throw "I/O error";
  for (i = 0; i < 8; i++)
  {
    fin >> aScore;
    fin.ignore(1000, 10);
    scoreTotal += aScore;
  } // for
  fin.close();

  // calculate and print the average
  average = scoreTotal / 8.0;
  cout << "The average of 8 numbers is “ << average << endl;
} // main
```

What if we wanted to modify the program to count the number of scores that are *greater than the average?*

Categories of data

- Numbers
- Text
- Graphics
- Audio
- Video

- Data categories are in order of how much storage they require
- There are certain efficiencies in the ways that computers deal with the simpler forms of numbers and text.

Variables

- A variable is something in a computer program that programmers use to store a value (in random access memory (RAM) during program execution).
- Data values can be specified in a program's code.
- Data values can be input from a keyboard or text file.
- Data values can be the result of a calculation.
- Variables are the things that store data values in a computer program.
- Variables are reusable.
- The number of variables that a programmer can use in a program is basically limited by the amount of memory in the computer, so for today's desktop computers, it is virtually unlimited.

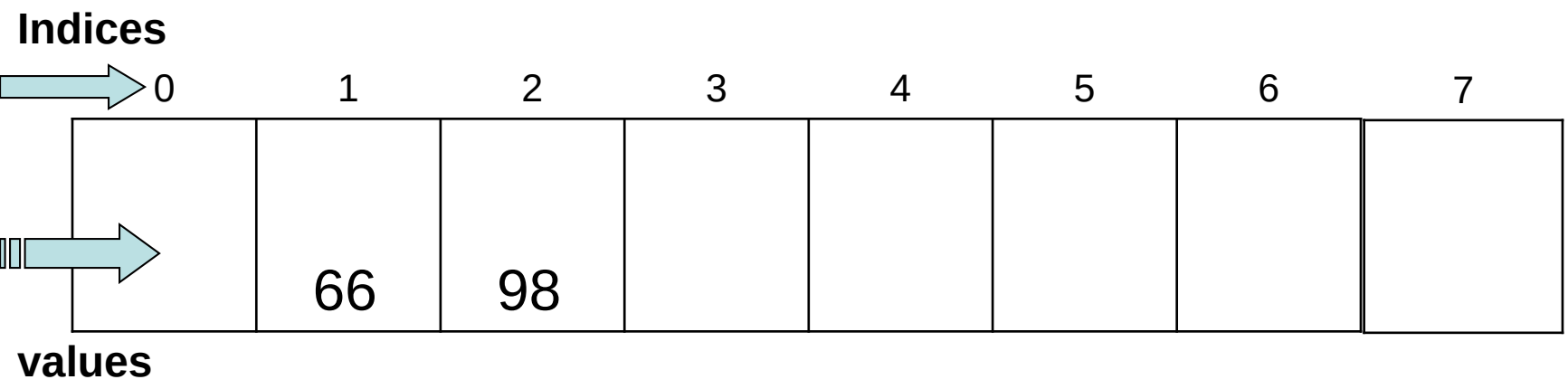
Visualizing Variables in RAM

Arrays

- Simple variables can also store more than one value, but whenever such a variable is set to a new value, the previous value is lost -- only the last value is ever remembered.
- An array is a variable that can hold more than one value by the same name.
- It uses an *index*, or integer number, to distinguish among the many values stored in it.

Visualizing memory: Array “score”

```
int score[8]; // declaration of array scores with 8 elements
score[1] = 66;
score[2] = 98;
```



Notice that the first element in an array has an index of 0, not 1.

Declaring an array

- This variable “score" can store 8 integer values!
- The first value is referenced as score[0] and the last as score[7], using "square brackets" around the index number.
- You say these as “score of zero" and “score of seven".
- Each of these is called an *element* of the array.
- The array index value can be expressed not only as a value, but also as a variable (or an expression that results in a integer).
- This means it can be used in a count-controlled loop, using the loop counter as an index (which is good for use with FOR loops).

Finding The Number Of Scores Greater Than The Average Algorithm

open 8Scores.txt for input
create an array to store 8 scores
8x read a score from the file
store the score in the array
close the input file
create scoreTotal and initialize to 0
8x retrieve a score from the array
add the score to scoreTotal
calculate and print the average score
create nGreater and initialize to 0
8x if score from the array is > average score
add 1 to nGreater
print nGreater with a label

avgFile2.cpp

How many times is the array score traversed?

```
…
#include <fstream>
#include <iostream>
using namespace std;
…
int main()
{
  //data
  ifstream fin; //represents input file
  int score[8];  //scores
  int i; // loop counter
  int scoreTotal = 0; //sum of scores
  double average = 0; //average score
  int nGreater = 0;
…
  // read and save the scores  fin.open("8Scores.txt");
  if (!fin.good()) throw "I/O error";
  for (i = 0; i < 8; i++)
  {
    fin >> score[i];
    fin.ignore(1000, 10);
  } // for
  fin.close();

  // calculate the average score
  for (i = 0; i < 8; i++)
  {
      scoreTotal += score[i];
  } // for
  average = scoreTotal / 8.0;
  cout << "The average of 8 numbers is " << average << endl;
  // count #of scores > average
  for (i = 0; i < 8; i++)
  {
     if (score[i] > average) nGreater++;
  } // for
  cout << nGreater << " scores are greater than the average." << endl;
} // main
```

Array size

- The array size equals the number of elements in the array. In our examples, the size of the array named "score" is 8.
- When the array is created, the values stored in those 8 elements are not predictable -- it is necessary to assign them before they are used.
- The data type for the elements of an array can be any valid C++ data type.

Array size

- The size of the array must be a literal integer, greater than zero.
- It cannot be a substituted with an integer variable.
- It is common for programmers to use *constants* in the declaration of a arrays.
- Constants are like variables, but they must be assigned upon declaration and they cannot be changed afterwards.

```
const int SIZE = 8;
int a[SIZE];
```

Constants

- Note that the name of the constant is fully uppercase, which is the naming convention for constants.
- Also note the use of the keyword **const** , denoting this as a constant instead of a variable.
- It is better to use a constant, rather than repeating the number "8" throughout the code.
- If we ever wanted to change the number of scores to be processed, it could be a lot of work to track down all the 8's, and that kind of editing is prone to error.

Array traversal

- The use of a "for loop" to process each element of an array is called *traversal*.
- This is the powerful thing about using arrays.
- Arrays let us store large numbers of values easily, and "for loops" let us process them easily.

Traversing An Array To Set Values Via The Keyboard

```
…
//data
const int SIZE = 8; //constant
int a[SIZE]; //array
int i; //loop index
…
//input data
for (i = 0; i < SIZE; i++)
{
  cout << "Please enter a[" << i << "]: ";
  cin >> a[i];
  cin.ignore(1000, 10);
} // for
```

Traversing An Array To Print Its Values

```
…
//data
const int SIZE = 8; //number of values
int a[SIZE]; //array of values
int i; //loop index
…
//output data
for (i = 0; i < SIZE; i++)
cout << "a[" << i << "] = " << a[i] << endl;
```

Curly brace initialization

- When arrays are created, the values of their elements are unpredictable.
- But it is possible to specify the values of each element upon declaration.
- It's called *curly brace initialization*

```
int daysPerMonth[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Curly brace initialization

- Note that the array size is not specified in the square brackets.
- Instead, C++ *counts* the number of elements and makes the array exactly that size.
- Actually, it's okay to specify the size in the square brackets anyway, as long as that number is equal to or greater than the number of elements in the curly braces.
- If the number is greater, the unlisted elements are set to zero.
- Curly brace initialization works only when an array is declared -- it cannot be used later to change the values in an array.

Array processing

- Arrays are processed by traversal.
- In traversal, you access each element of the array, one-by-one, for assignment, output, or any other operation.
- Some more advanced types of array processing involve searching and sorting.

A Counting Search Loop

```
…
//data
int nGreater = 0; //number of values greater than the average
…
for (i = 0; i < SIZE; i++)
{
  if (score[i] > average) nGreater++;
}//for
```

- The variable **nGreater** is called an *accumulator*. It is a variable that gets set to an assumed value *before* the search loop starts, and gets adjusted inside the loop.
- **scoreTotal** in the same example is also an accumulator -- it is assumed to be zero, and adjusted as each element is inspected.

2 Boolean search loops

```
//data
bool hasPerfectScore = false;
int i; //loop index
…
//check for a perfect score
for (i = 0; i < SIZE; i++)
{
  if (score[i] == 100)
  {
     hasPerfectScore = true;
  } // if
} // for
```

Finding the maximum value

```
…
//data
int max = score[0]; //maximum score
…
//find maximum score
for (i = 1; i < SIZE; i++)
{
  if (max < score[i]) max = score[i];
}//for
```

Finding the minimum value

```
…
//data
int min = score[0]; //minimum score
…
//find minimum score
for (i = 1; i < SIZE; i++)
{
  if (min > score[i]) min = score[i];
}//for
```

Find the maximum and minimum values

```
…
//data
int max = score[0];
int min = score[0];

…
//find minimum and maxium
for (i = 1; i < SIZE; i++)
{
  if (max < score[i]) max = score[i];
  if (min > score[i]) min = score[i];
} // for
```

Sorting

- There are many methods to do sorting of values.
- In this course we will only learn how to do the “bubble sort” method.
- Sorting methods differ by efficiency.

See example program

stringSort.cpp

Sorting

- The nested loops sort the text alphabetically, because of the statement **if (day[i] > day[j])** .
- To sort in reverse alphabetical order, use **<** instead of **>** .
- In comparing text, remember that case matters.
- In the ASCII sequence, uppercase letters come before lowercase.
- So the word "Zebra" actually comes before "aardvark", since 'Z' is ASCII code 90, and 'a' is code 97.
- To make a case independent comparison, you could create uppercase versions of "day[i]" and "day[j]", and compare them in the "if" statement.

Text length comparisons

- You could also compare based on text length instead of alphabetically.
- To do so, you would change the "if" statement like this:

if (day[i].length() > day[j].length())

Dynamically sized arrays

- C++ has another way to declare arrays, which allows the number of elements to be specified by a variable.
- Here is the statement to declare an array of "n" elements, where "n" is an integer variable:

int* score = new int[n];

delete [ ] score;

- This form of array declaration does not allow for curly brace initialization.

Dynamically sized arrays

- Data type "int star“ is used to store "memory addresses" -- that is, it stores the number of the first byte in memory used for the array.
- The variable "score", so declared, is called a *pointer*, because it indicates where in memory that the array is stored.

- Example: dynamicArray.cpp

Arrays in Function parameter lists

- Array values can be shared with functions by using the parameter list, much like other variables.
- When variables are shared with functions, copies are made.
- So if the copy gets changed in the function, it does not affect the original.
- This called *pass by value*.

Arrays in Function parameter lists

- But when an array's values are shared with a function, no copies are made -- instead, the original array is shared.
- Any changes made to the elements of the array inside the function *do* affect the original array!
- This called *pass by reference*.

Arrays in Function parameter lists

- To specify an array in the parameter list, use a data type that matches the elements of the array, but *append* a ***** .
- Yes, that makes it a pointer -- it contains the memory address where the array exists so that the function can get to the original.
- Example: an array of integers is shared with a function whose job is to return the average of the elements in the array

Arrays and functions together

- To show how arrays and functions work together, let's rewrite **avgFile2.cpp** with functions.
- One of the functions will be used to read the file, fill the array, and send the array back to "main".
- In so doing, the details of getting values for the array are left to a *subprogram*, as are the details of printing and averaging the values.

- Example avgFile3.cpp

Global Constants

const int SIZE = 8;

- It is okay to declare constants above the function definitions -- what it does is to allow all functions to have access to the constant, so that it does not have to be repeated.
- Any constant or variable declared inside a code block is no longer available after the first closing curly brace to its left.
- This gets around that limitation -- it's called a *global constant*.
- In general, you can declare constants either inside functions or globally -- it's the programmer's choice.
- The advantage of global is that it can be shared among functions.
- If only one function uses the constant, there is no advantage to making it global, but it is okay to do so anyway.

Global Constants

- Do not be tempted to make variables global!
- It is possible to do so, and the compiler is okay with it, but it is bad programming practice.
- Why is it okay for constants but not for variables? It's a code maintenance issue.
- Because constants do not change, there is really no problem making them global and using them throughout.
- But a global variable can be reassigned anywhere, making it hard to track in a large program.
- Plus it precludes having functions that call themselves (recursion).

Local variables in Functions

- Note that "i" is declared separately in every function except "main".
- This is because it is needed for the loops in each of these functions, and while there are ways to share one counter variable among all, it is better for them to each have their own.
- The identifier "i" is used multiple times, which is not allowed within the same scope, but these are separate scopes.
- Variables that are declared inside a function are called *local variables*, because of their "local" scope.

Parameter List Variable names

- The second parameter in "countScoresGreater" is named "x" in the parameter list, but the call in "main“ has a variable named "average" in that position.
- The names do not match, and they do not have to.
- Whether the names refer to separate copies of an integer or a floating point value, or it they refer to the same array, the names are local to their functions.
- In the same way that a person can go by two different names -- one at work and another at home -- variables can, too.

Temporary variables

- The variable "average" in "main" is used to store the result of the "getAverage" call.
- It is really not needed, though, because "average" could be replaced where it appears in the two output statements with **getAverage(score)** .
- But doing so would require that the averaging loop be executed twice, which is a bit of a waste.
- So programmers generally will use an extra "temporary" variable to store the result of a call that is used multiple times.
- Note that this was not done with "countScoresGreater", because it is only called once.

Pass by reference

- In "readScores", array elements are assigned values.
- Since arrays are *passed by reference*, the function gets the *original* version of the array, and any changes made to it affect "main" where it was declared.
- This would not be so for anything that is not an array -- integers and doubles are *passed by value*, meaning that the function has a *copy* of the original value.
- Any change made to it in the function does not affect the original in "main".

How to share an ENTIRE ARRAY with a function

```
void fun(int* a) // the * indicates that "a" is an array
{
...
}
...
int main()
{
  int a[10];
...
  fun(a); // the "a" parameter sends the whole array to the function
...
}
```

How to share a SINGLE ELEMENT OF AN ARRAY with a function

```
void fun(int x)
// the function does not know or care that "x" lives in an array
{
...
}
...
int main()
{
  //data
  int a[10]; //array of values
...
  fun(a[0]);
// the parameter refers to ONE ELEMENT of the "a" array
...
}
```

4 ways to create arrays

1. //fixed size array
   int scores[8];
2. //using constant to represent number of elements in array
   const int SIZE=8;
   int scores[SIZE];
3. // { } initialization
   int scores[] = {66, 99, 52, 100, 0, 75, 77, 90}
4. int n = 25; //represents number of elements in array
   int* scores = new int[n]; //dynamic array allocation

References

- INTRODUCTION TO PROGRAMMING USING C++ by Robert Burns