CSCI 240 Lecture Notes - Part 7

Arrays - Part 1

Up to now, we have used simple data types (int, char, etc.) except for strings. Now we look at the first ***data structure***. A data structure is a **compound** data type; that is, it is a single entity made up of numerous parts.

It's kind of like a **numbered list** of "things" where every "thing" must be of the same data type - all ints or all chars, etc...

Arrays can help solve problems that would otherwise be quite difficult and can sometimes make solutions clearer.

As an example of a problem that is difficult without arrays, consider the following:

Problem: read n values and calculate an average. For each number, print whether it is above, below, or equal to the average.

The first part of the problem is easy to solve. Values are read from the user and then the average is calculated. The second part isn't so easy because in the process of reading the values from the user we "lose" the previous values and therefore don't have anything to compare with the average except for the last value entered.

Solution 1

If we know there are exactly *n* numbers, we can declare *n*variables to hold all of the values. So if there are 3 numbers:

```
int n1, n2, n3;
double avg;

cout << "Enter the first number: ";
cin >> n1;

cout << "Enter the second number: ";
cin >> n2;

cout << "Enter the third number: ";
cin >> n3;

avg = (n1 + n2 + n3) / 3.0;

if (n1 > avg)
  {
  cout << "n1 is above";
  }
else if (n1 < avg)
  {
  cout << "n1 is below";
  }
else
  {
  cout << "n1 is equal to avg";
  }

//same pattern of code repeated for n2 and n3
```

Problem: what if there are 50 or 500 or 5000 numbers? This solution, awkward when n = 3, is terrible when n get big.

Solution 2

Read the numbers twice. That is: read in all the numbers and calculate average, read in all the numbers again, this time checking each as it is read against the previously calculated average.

If input is from keyboard, then the user has to enter each number twice. Accurately, with no mistakes.

If input is from an *explicitly opened file* (which we will be able to do later), it will work, but is inefficient because the file has to be read twice.

Note: this will not work with *I/O redirection* - we can't re-open the file and read from the beginning again.

Solution 3:

Use arrays to *store* all the values as they are read and thus "remember" them.

First some information on how to create and manipulate arrays.

An ***array*** is a data structure consisting of an ordered set of data values of the same type. Think of it as a numbered list, but all items in the list must be the same kind of thing (the same data type).

When defining an array in a program, we have to specify three things:

- what kind of data it can hold (ints, chars, doubles, strings, etc.)
- a name
- how many values it can hold (i.e. the maximum number it can hold)

The generic format for declaring an array:

```
data_type array_name[ # of items to hold ];
```

So, to declare an array that can hold up to 10 integers:

```
int numArray[10];
```

This reserves *space* for 10 integers. It does ***NOT*** *place* ***any*** *values* into the array.

Each position in the array is called an ***array element***.

The elements in this array are numbered from 0 to 9. (**Not** 1 to 10.)

Notice that when you declare an array, you specify **how many items** it can hold. And that these elements are numbered starting at 0 and ending at one less than the number of elements. This array holds 10 integers, which are numbered from 0 to 9.

You must use a literal number or a previously declared *const int* or #defined constant in an array declaration. No regular variables.

In our code when we need to refer to a particular element in an array, we use ***subscript*** notation: [n]

n can be:

- an integer literal: [3]
- an integer variable: [n]
- an integer expression: [n+1]
- any scalar expression: ['A'] (= to [65] //ASCII value of 'A')
- any function that returns an integer value

NOTE: that in this example, the ['A'] example refers to a non-existent array element - since 'A' is 65 and the array only has room for elements up to [9]. This would almost certainly result in a program bug. If we had declared numArray to hold 100 values, it would be perfectly fine).

Note that [n] is never used by itself. It is used to specify **which** array **element** to refer to, but we also need to specify **which array**:

So to refer to the value of a particular array element, we write:

```
array_name[ arrayelement # ]
```

So, for example, to assign values to (i.e. store values into) an array:

```
numArray[0] = 1;
numArray[1] = 3;
numArray[2] = 5;
...
numArray[9] = 19;
```

Or:

```
for (i = 0; i < 10; i++)
  {
  numArray[i] = (i*2) + 1;
  }
```

Notice how the variable *i*, used as a subscript, runs from 0 to "less than" 10, i.e. from 0 to 9.

To **get** values from an array, the same notation is used:

```
cout << numArray[2];

thirdOdd = numArray[2];

nthOddSquared = numArray[n-1] * numArray[n-1];
```

Example: To add up the 1st ten elements in an array:

```
sum = 0;

for (i = 0; i < 10; i++)
  {
  sum += numArray[i];
  }
```

Remember:

- [n] specifies which position you are referring to
- numArray[n] evaluates to the value stored in the array at position n
- elements are numbered starting at 0
- when you declare an array to hold n elements, its subscripts go from 0 to n-1

Now we can solve the problem posed earlier. Assume that a negative number is the end-of-data signal.

Solution 3 code

```
#include <iostream>
#include <iomanip>

using namespace std;

const int ARSIZE = 100;

int main()
{
int numArray[ARSIZE];      //Note use of constant

int numberElements = 0,    //number of elements in the array
    i,                     //subscript for processing the array
    num,                   //number from the user
    sum = 0;               //sum of the numbers

double avg;

//read numbers from the user and put them into the array

cout << "Enter an integer number (negative to quit): ";
cin >> num;

while ( num >= 0 )
  {
  numArray[numberElements] = num;

  numberElements++;

  cout << "Enter an integer number (negative to quit): ";
  cin >> num;
  }


//Add up elements. Note that numberElements is the total number of values that
//were entered by the user before the negative number

for ( i = 0; i < numberElements; i++ )
  {
  sum += numArray[i];
  }


//Calculate the average of the numbers

avg = (double) sum / numberElements;


//Now go through array and print whether the number is < > = the average

for ( i = 0; i < numberElements; i++ )
  {
  if ( numArray[i] < avg )
    {
    cout << numArray[i] << " is less than the average of " << avg << endl;
    }
  else if ( numArray[i] > avg )
    {
    cout << numArray[i] << " is greater than the average of " << avg << endl;
    }
  else
    {
    cout << numArray[i] << " is equal to the average of " << avg << endl;
    }
  }


return 0;
}
```

More Notes on arrays:

Arrays can be initialized when they are declared:

```
int ar[5] = { 1, 3, 5, 7, 9 };            //exactly enough values

int ar2[5] = {2, 4, 6};                    //the rest are 0

int ar3[] = {1, 4, 9, 16};                 //allocates 4 elements at positions 0..3

char vowels[] = {'a', 'e', 'i', 'o', 'u'};      //5 elements

double sqrts[4] = {1.0, 1.414, 1.732, 2.0};
```

Executable program statements, such as ones that assign computed values, or data read from the keyboard or a disk file can also be used to initialize an array.

Just because an array was declared to hold a specific number of values, the entire array does not have to be used. It's typical to declare an array to hold more values than are expected and have the program keep track of how many array elements are actually used as values are put into the array. This is what happened in Solution 3.

Sometimes it is more natural to start at 1, so you could just not use ar[0]:

```
int monthlenAr[13] = {0, 31, 28, 31, ...};
```

Then for January, use monthlenAr[1], for February use [2], etc.

Other Misc. Operations

To increment the ith element:

```
ar[i]++;

ar[i] += 1;

ar[i] = ar[i] + 1;
```

To add n to the ith element:

```
ar[i] += n;

ar[i] = ar[i] + n;
```

To copy the contents of the ith element to the kth element:

```
ar[k] = ar[i];
```

To exchange the values in ar[i] and ar[j]:

First, understand that you must declare a "temporary" variable to hold one value, and that it should be the same data type as the array elements being swapped:

```
int temp;

temp  = ar[i];        //save a copy of value in i
ar[i] = ar[j];        //copy value from j to i
ar[j] = temp;         //copy saved value from i to j
```

Sample Exercises:

1. Given a character in variable *ch*, determine if it is a vowel, i.e. if *ch* matches one of the values in:

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

2. An array of 10 elements called AR exists. Find and print the smallest element in it.

3. A Fibonnacci sequence consists of a series of integers in which each is equal to the sum of the of the previous two numbers. The first two values are 1 and 1. So the series is: 1, 1, 2, 3, 5, 8, 13 ...

   Write a program to store the first 50 Fibonnaci numbers in an array using executable statements (ie. don't initialize the array when it's declared). Then fill a second array with the squares of these numbers.

4. Write a program to read numbers from the keyboard until the user enters a negative number. Once all of the numbers have been entered, print them out in reverse order.