Arrays

http://people.sc.fsu.edu/~jburkardt/isc/week07_lecture_13.pdf

.....

ISC3313:
Introduction to Scientific Computing with C++
Summer Semester 2011

.....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 21 June 2011

Arrays

- **Introduction**
- Arrays in C++
- Indexes and FOR Loops
- Counting Dice Throws
- Search an Array
- Lab Exercise #7

INTRO: Schedule

Read:

- Today's class covers Sections 6.1, 6.2, 6.3, 6.4, 6.6

Next Class:

- Sort, Mathematical Vectors, 2D Arrays

Assignment:

- Today: in-class lab exercise #7
- Thursday: Programming Assignment #5 is due.

Midterm Exam:

- Thursday, June 30th

INTRO: Schedule

So far, we have been dealing with simple problems and small amounts of data. Each item of data was stored as a separate varaible, with its own name, type and value. However, a problem doesn't have to get very large before it becomes extremely tedious to program this way. Even in the simple dice counting problems, we had to do the same thing six times, depending on whether the result was a 1, 2, 3, 4, 5 or 6.

If data can include thousands of values (it can) and programmers can write programs that read in such data and work with it (they do) then how do they avoid having to think up thousands of different names for variables, and carrying out a thousand computations using a thousand lines of code?

INTRO: Schedule

C++ allows a program to create an **array** of data, which groups many values under a single logical name. Each value in the array can be located using a numerical index. With this simple idea, programs for handling enormous amounts of data become short and concise.

Today we will look at the rules for creating and using an array, and consider some simple operations that can be carried out.

Arrays

- Introduction
- **Arrays in C++**
- Indexes and FOR Loops
- Counting Dice Throws
- Search an Array
- Lab Exercise #7

ARRAY: One Variable, Many Values

So far, each single number we want to store in C++ requires a separate variable, that is, a name and a type, in which we can store a single value.

Especially for scientific calculations, it is very natural to think of a collection of data as a unit. For instance, we might have made measurements of soil density at 100 locations. We might have recorded the distance between Mars and the Earth every day for a year, or the attendance at each FSU football game in 2010.

When we have data involving many examples of the same kind of object, C++ allows us to store them as a sort of list, called an **array**. An array is a new kind of variable, which, along with a name and type, now has an extra feature called its **dimensio**
size, indicating how many single values it can store.

ARRAY: Declaring an Array

Because an array is a variable, it must be declared before it is used. The declaration must include the dimension, following the array name in square brackets.

Thus, 100 densities, stored as a **float**, or 365 planetary distances, stored as a **double**, or 13 attendance records, stored as an **int**, might be declared as:

```
float density[100];
double distance[365];
int attendance[13];
```

Thus, the dimension of the **distance** array is 365. It has room for 365 entries.

ARRAY: Declaration With Initial Values

When you declare an array, you can list the starting values of its entries, just as you can for a regular variable. Because the set of values is a list, you must enclose them in curly brackets.

```
float areas[4] = { 47.25, 49.80, 37.50, 43.75 };
int coins[5] = { 1, 5, 10, 25, 50, 100 };
```

ARRAY: Declaration With Initial Values

An array contains the "starting address" for the values it contains. The first value occurs immediately. The second value is one "unit" later, and so on.

```
Name   Index   Value
-----  -----   -----
areas    0     47.25   <-- First entry in areas
         1     49.80
         2     37.50
         3     43.75
coins    0       1     <-- First entry in coins
         1       5
         2      10
         3      25
         4      50
         5     100
```

ARRAY: Referring to a Single Value

An array is one variable that contains many values. Normally, when we carry out a computation, we want to refer to one particular value.

To refer to a particular element of an array, you must specify the array name followed by the index of the variable, enclosed in square brackets. The index system in C++ starts at 0. So the first entry stored in array **distance** has index 0, and is referred to as **distance[0]**, followed by **distance[1]** and so on up to **distance[364]**.

Remember, an array **a** declared with the statement **int a[100]** contains entries **a[0]**, **a[1]**, ..., **a[99]**.

ARRAY: Referring to a Single Value

An array entry, specified by an array name with an index value, can be used on the left or right hand side of a formula, just like any other variable of its type.

```
float density[100];

weight      = density[5] * volume;
density[7]  = density[3] * 2;
density[99] = sqrt ( density[98] );
cout << "Density[43] = " << density[43] << "\n";
if ( density[10] < density[11] )
{
  density[10] = density[10] + 1.0;
}
```

For an array of dimension 100, the array index should be between 0 and 99. Any other index value is called an *out of range* index, and represents a programming mistake. However, C++ does not check for such errors or warn about them.

ARRAY: Advantages of Arrays

When we were counting the results of tossing a die, I mentioned that this task would be easier when we learned about arrays.

However, from what I have told you so far, the only advantage is that, instead of having to declare six variables and initialize them, I only have to make one declaration. In other words, this:

```
int count1 = 0, count2 = 0, count3 = 0, count4 = 0, count5 = 0, count6 = 0;
```

becomes

```
int count[6] = {0,0,0,0,0,0};
```

Of course, this is better, but if I work with cards, I will still have to type 0 52 times. This means I would still not want to work with an array like this, containing 1,000 entries to be initialized!

ARRAY: Advantages of Arrays

The more serious problem is that, in counting the die results, I used a switch statement. That switch statement will still be just as long as before, and the only thing that changes is the variable names:

```
switch ( value )
{
  case 1:
    count1 = count1 + 1;   <-- replace by:   count[0] = count[0] + 1;
    break;
  case 2:
    count2 = count2 + 1;   <-- replace by:   count[1] = count[1] + 1;
    break;
    ...
  case 6;
    count6 = count6 + 1;   <-- replace by:   count[5] = count[5] + 1;
    break;
  default;
    break;
}
```

I was free to use **count1** for the name of the variable counting the result of 1, but using a single array, I have to follow the indexing rules, so it becomes the "zero-th" entry of array **count**.

Arrays

- Introduction
- Arrays in C++
- **Indexes and FOR Loops**
- Counting Dice Throws
- Search an Array
- Lab Exercise #7

FOR: Initialization

The key to using arrays efficiently is to refer to their entries using a **for** loop to generate the indexes. It's fine to initialize an array of size 6 to zero by listing the values:

```
int count[6] = {0,0,0,0,0,0};
```

but this becomes impractical for an array of size 1,000. In such a case, however, we can simply use a **for** loop to set the entry in **count[i]** to zero.

```
int count[1000], i;

for ( i = 0; i < 1000; i++ )
{
  count[i] = 0;
}
```

If the array changes size, we only change the limit on the loop

FOR: Indexes Range from 0 to N-1

When using a **for** loop, if the array has dimension **n**, then the entries in the array have indexes from 0 to **n-1**.

A typical **for** loop that indexed every entry might be written

```
for ( i = 0; i < n; i++ )
{
...
}
```

or

```
for ( i = 0; i <= n - 1; i++ )
{
...
}
```

FOR: Assignment Statements

If the entries in the array can be computed by some kind of formula, then a **for** loop can take care of it.

Here, we compute angles from 0 to 90 degrees, and create arrays to hold the values of the sine and cosine:

```
double c[91], s[91], pi = 3.14159265, angle;
int i;

for ( i = 0; i <= 90; i++ )
{
  angle = ( double ) i * pi / 180;
  c[i] = cos ( x );
  s[i] = sin ( x );
}
```

Here, we make a table of the Fibonacci numbers:

```
int f[50], i;

f[0] = 1;
f[1] = 1;
for ( i = 2; i < 50; i++ )
{
  f[i] = f[i-1] + f[i-2];
}
```

FOR: Printing

If the entries in the array can be printed on one line, you can use a simple **for** loop, remembering to put a space before each value, and an end line character at the end:

```
double areas[4]
int i;

for ( i = 0; i < 4; i++ )
{
  cout << "  " << areas[i];
}
cout << "\n";
```

For a longer array, you might print one value per line, but in that case you might include the index information as well:

```
float trig[91];
int j;

for ( j = 0; j <= 90; j++ )
{
  cout << "  " << j << ": " << trig[j] << "\n";
}
```

FOR: Interactive Entry of Array Values

read_array.cpp: If you wish the user to enter the values of an array interactively, you can use a **for** loop:

```
# include <cstdlib>
# include <iostream>

using namespace std;

int main ( )
{
  int i;
  int list[5];

  cout << "Enter the 5 entries of the array LIST: ";
  for ( i = 0; i < 5; i++ )
  {
    cin >> list[i];
    cout << "  " << i << ":  " << list[i] << "\n";
  }
  return 0;
}
```

FOR: Using Array Entries in Formulas

Every entry of an array is a variable; you can use any indexed array entry in a formula or assignment statement.

Suppose I have pennies, nickels, dimes, and quarters in my bank, and let the array **n**, of dimension 4, store the number of coins of each type, and **v** their value.

```
int n[4] = { 17, 4, 13,  6 };
int v[4] = {  1, 5, 10, 25 };
```

The number of dimes I have is n[2], the value of one dime is v[2], and the total value of dimes I have is:

```
dval = n[2] * v[2];
```

The total number of coins I have is n[0] + n[1] + n[2] + n[3]
How do I express the total value of my coins:

```
total_val = ?
```

FOR: Using Array Entries in Formulas

Suppose we had a machine that would accept 5 pennies, and return a nickel. If we wanted to use that machine, how would it affect our penny and nickel counts?

Since **n[0]** is the number of pennies we have currently, dividing by 5 will give us the number of nickels we can form. The remainder is the number of pennies we are stuck with.

To express the change in our situation, we need to add the new nickels to the old ones, and reset the penny count to the remainder:

```
convert = n[0] / 5;
keep = n[0] % 5;
n[1] = n[1] + convert;
n[0] = keep;
```

(If you keep trying to convert your change upward, you will see that when you get to quarters, things become a little more complicated, because you can't form a quarter using only dimes.)

Arrays

- Introduction
- Arrays in C++
- Indexes and FOR Loops
- **Counting Dice Throws**
- Search an Array
- Lab Exercise #7

DICE: Storing Dice Results

Last week, we had an example in which we were modeling the behavior of dice. I simulated the throw of a single die, getting a value between 1 and 6. For every possible result, I had to name a variable, and have a separate section of a **switch** statement that incremented that variable when that result occurred.

Thus, if I got a 4, there was a variable called **count4**, and my **switch** statement included the lines:

```
    break;
  case 4:
    count4 = count4 + 1;
    break;
  case 5:
    count5 = count5 + 1;
    break;
```

...and so on...

DICE: Array Setup for One Die

It seems as though the word **count** is a variable, and I am keeping track of 6 different numbers by slightly modifying the name each time. Suppose, instead, I use a single array, called **count**, of dimension 6, to keep track of the results.

My data naturally runs from 1 to 6, which the array indices run from 0 to 5. So I will want to store information about the score 1 in index 0, score 2 in index 1, and so on:

```
count[0]  <-- data for "1"
count[1]  <-- data for "2"
count[2]  <-- data for "3"
count[3]  <-- data for "4"
count[4]  <-- data for "5"
count[5]  <-- data for "6"
```

DICE: Array Setup for One Die

Notice that while our scores run from 1 to 6, the locations or indexes we use from from 0 to 5. It might seem like C++ is trying to make life difficult for us!

However, we can try to avoid confusion by using two variables:

1. **score** keeps track of scores. It goes from 1 to 6;
2. **i** keeps track of array indexes. It goes from 0 to 5.

and now remember the following rules:

1. data for a given **score** goes in index **i=score-1**;
2. the index **i** stores data for **score=i+1**.

So the score "3" is stored in index 2. Index 5 stores scores of 6.

DICE: Program for One Die

Now simulating many rolls of a die becomes simpler:

```
int count[6], i, score;

for ( score = 1; score <= 6; score++ )    <--Initialize
{
  i = score - 1;
  count[i] = 0;
}

for ( j = 1; j <= n; j++ )                <-- Roll and Count
{
  score = random_int ( 1, 6 );
  i = score - 1;
  count[i] = count[i] + 1;
}

for ( score = 1; score <= 6; score++ )    <-- Print
{
  i = score - 1;
  cout << score << "  " << count[i] << "\n";
}
```

DICE: Array Setup for Two Dice

Now suppose we wanted to count the occurrences of each score using *two* dice? Only a few simple changes are needed.

When rolling two dice, there are 11 possible scores, ranging from 2 to 12. So we have to think of how we are going to store this data in an array. One way sets up an array of size 11, and puts the first score (2) in the first entry (index 0), and so on:

```
count[0]  <-- Data for "2"
count[1]  <-- Data for "3"
count[2]  <-- Data for "4"
...
count[10] <-- Data for "12"
```

Data for **score** ends up in **count[i]**, where **i** = **score** - 2.

DICE: Program for Rolling Two Dice

```
int count[11], i, score;

for ( score = 2; score <= 12; score++ )                <--Initialize
{
  i = score - 2;
  count[i] = 0;
}

for ( j = 1; j <= n; j++ )                             <-- Roll and Count
{
  score = random_int ( 1, 6 ) + random_int ( 1, 6 );   <-- Two dice
  i = score - 2;
  count[i] = count[i] + 1;
}

for ( score = 2; score <= 12; score++ )                <-- Print
{
  i = score - 2;
  cout << score << "   " << count[i] << "\n";
}
```

DICE: Array Setup for Three Dice

The point that needs to be made here is that you should now see that if someone told us we had to model 3 dice, for 100,000 throws, it is very easy to do.

We adjust the value of **n** to 100,000;

We add one more + **random_int (1, 6)** to simulate the third die;

We determine the range of possible scores: 3 to 18, which is 16 possible values;

We expand the dimension of **count** to 16;

We work out that the index **i** is computed by **i = score - 3**;

This is only possible because we are using arrays.

Arrays

- Introduction
- Arrays in C++
- Indexes and FOR Loops
- Counting Dice Throws
- **Search an Array**
- Lab Exercise #7

SEARCH: Examine Every Value

If we have a list of data, many tasks become very simple by using an array for the data, and a **for** loop to examine each item.

If we have a list of 100 integers, how would we determine:

- whether the list contains the value 17;
- the sum of the entries;
- the average of the entries;
- the largest entry;
- whether the list is sorted in increasing order;
- a random entry of the list;

SEARCH: Find Entry Equal to 17

For instance, we might ask if an entry 17 occurs, or where such an entry occurs, or how many times it does.

```
bool occurs = false;
int i, where = -1, count = 0;
int list[100];

for ( i = 0; i < 100; i++ )
{
  if ( list[i] == 17 )
  {
    occurs = true; |  where = i; |  count = count + 1;
    break;         |  break;     |
  }
}
```

I have illegally stuck all three sets of code in one **for** loop.
An actual program would only use one set.

SEARCH: Sum or Average

Computing the sum or average simply requires a running total.

```
int i, sum = 0;
float average;
int list[100];

for ( i = 0; i < 100; i++ )
{
  sum = sum + list[i];
}
average = ( float ) sum / 100.0;
```

SEARCH: Maximum

When seeking the maximum, you can initialize to a very negative number (not a great idea), or to the first array entry (better).

```
int list_max;
int list[100];

list_max = -100000000;     <-- Simple, slightly dangerous
list_max = list[0];        <-- Always works.
for ( i = 0; i < 100; i++ )
{
  if ( list_max < list[i] )
  {
    list_max = list[i];
  }
}
```

If you take the second approach, your loop can be slightly more efficient by starting at $\mathbf{i = 1}$.

SEARCH: Random

To select a random entry from a list, we select a random index.

```
int i, index, list[10000], samples = 20, sum;
float average;

sum = 0;
for ( i = 1; i <= samples; i++ )
{
  index = random_int ( 0, 9999 );  <-- 9999, not 10000!
  sum = sum + list[index];
}
average = ( float ) sum / ( float ) samples;
```

Here, I'm trying to estimate the average of a list by randomly samplying just 20 items.

SEARCH: Sorted?

An array is sorted if every consecutive pair of numbers is sorted.

```
int i, list[100];
bool sorted;                          <-- A bool variable can
                                          store a logical value
sorted = true;
for ( i = 0; i <= 98; i++ ) <-- Do 99 checks, not 100
{
  if ( list[i] > list[i+1] )<-- Something out of order
  {
    sorted = false;
    break;                            <-- No need to check more
  }
}
if ( sorted )
{
  cout << "The array is in sorted order!\n";
}
```

Arrays

- Introduction
- Arrays in C++
- Indexes and FOR Loops
- Counting Dice Throws
- Search an Array
- **Lab Exercise #7**

EXERCISE: A Histogram of the Mona Lisa

The file **mona.pgm** is a PGM graphics file containing a gray scale image of the Mona Lisa. The file is actually a text file, which means you can open it with an editor to see the data; on the other hand, you can view the image by typing

```
eog mona.pgm
```

The file begins with four special lines of data:

```
P2          <-- Indicates this is a PGM file
250 360     <-- The image has 250 columns and 360 rows
255         <-- The white value is 255 (maximum)
```

The rest of the file is 250x360 integers between 0 and 255, representing the gray value of each pixel.

EXERCISE: Group the Data into 16 Ranges

Your program will read the image data, one gray value at a time, and produce a summary of the range of grays. It would be easy to do this using an array of 256 entries, one for each possible value. But this gives too much data.

Instead, you should set up 16 counters, having the following ranges:

```
counter[0]:     0 <= G < 16 = 1 * 16
counter[1]:    16 <= G < 32 = 2 * 16
counter[2]:    32 <= G < 48 = 3 * 16
...
counter[15]: 240 <= G < 256 = 16 * 16
```

It is your responsibility to declare the array, initialize it, update it as each new value of G is read, and to print the value of the array at the end.

EXERCISE: Quickly Determine the Counter!

Please note that it is possible, but very painful and inefficient, to determine which counter entry to update by using 16 **if/else** statements. You will not get full credit if you do things this way!

Instead, think about the relationship between the range and the corresponding counter index.

Why does **counter[2]** correspond to a range of $32 \leq G < 48$?
Why does **counter[10]** correspond to a range of $160 \leq G <= 176$?

If G is 57, why do I know *immediately* that it updates counter 3?

If you think about what is going on, you will be able to write a single statement that tells you what counter to update.

EXERCISE: A Histogram of 16 Gray Ranges

The web page has the graphics file **mona.pgm** and the partial program **mona.cpp**, which will read the image file for you.

You need to modify the program in three places:

1. declare a counter array, and initialize it;
2. given a value G, determine which counter entry to update;
3. print the counter array at the end.

Once your program has printed its results, please show your work to Detelina so you can get credit for the exercise!