Sorting, Polynomials

http://people.sc.fsu.edu/~jburkardt/isc/week07_lecture_14.pdf

.....

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

.....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 26 June 2011

Sorting, Polynomials

- **Introduction**
- Bubble Sorting
- Insertion Sorting
- Sending an Array to a Function
- Polynomials
- Assignment #6

INTRO: Schedule

Read:

- Today's class covers information in Sections 6.5, 6.7, 6.8

Next Class:

- 2D Arrays, Strings

Assignment:

- Programming Assignment #5 is due today.
- Programming Assignment #6 will be due July 7.

Midterm Exam:

- Thursday, June 30th

INTRO: Introduction

We will continue our discussion of C++ arrays.

We start by looking at how a numeric array can be **sorted**. One simple method assumes we have all the numbers available at the beginning. The second method can be used even in the case where the entries in the array become available one at a time, as though someone were dealing us a hand of cards.

We then look at how arrays can be passed to a function, and notice something surprising about how data changes occurring inside a function may or may not be returned to the calling program.

We take a look at the mathematical topic of polynomials, and consider how some useful operations involving polynomials ca implemented in C++.

Sorting, Polynomials

- Introduction
- **Bubble Sorting**
- Insertion Sorting
- Sending an Array to a Function
- Polynomials
- Assignment #6

BUBBLE: Sort One Thing at a Time

When we say an array is sorted, we usually mean that the entries are listed in increasing order.

Often, a procedure to be carried out on data which be much faster if the data is sorted. And that means that we need to find methods that will help us sort an arbitrary list.

A simple sorting method is known as *bubble sort*. The name comes from the idea that, as the elements are sorted, they rise through the array like bubbles, switching places with other elements.

BUBBLE: Compare Neighbors

Let us assume that we have an array **list[]** of **n** numeric elements, to be sorted.

It's hard to know how to approach the problem of sorting. But remember that last time, we were able to check whether an array was sorted simply by comparing neighboring values.

Suppose we go through the array, comparing neighboring values, but this time, if we find a discrepancy, *we do something about it*, namely, we interchange the two values.

This probably won't make the whole array sorted, but it will make it a little more sorted than it was.

BUBBLE: How Do We Swap Two Items?

It's hard to believe, but even a simple operation like interchanging or swapping two numbers involves an extra step that our minds don't think about.

Suppose we have two variables, called **a** and **b**, which have the values 17 and 2, and I want to swap them, so that they are sorted, that is, so that $a < b$. What's wrong with the following code?

```
int a = 17, b = 2;


a = b;
b = a;
```

Oddly enough, this does not produce a situation in which **a** = 2 and **b** = 17. Instead, both values are now 2.

BUBBLE: How Do We Swap Two Items?

In order to swap two items, we have to make use of a temporary variable. It's like an extra parking place that we need when interchanging the locations of two cars.

We need some place to park the value of **a** before we move the value of **b** there. Then we can go back to where we parked **a** and move it into the empty spot left by **b**. A temporary variable is often named **t** or **temp**:

```
t = a;
a = b;
b = t;
```

To interchange elements **list[i]** and **list[i+1]** of an array, we write:

```
t = list[i];
list[i] = list[i+1];
list[i+1] = t;
```

BUBBLE: Bubble Sort, One Sweep

Compare neighbors, and swap if necessary:

```
        0 1 2 3 4 5 6
--------------------
      | 8 7 4 9 0 6 2  <-- Before the first sweep.
0<1?  | 8 7 4 9 0 6 2
1<2?  | 7 8 4 9 0 6 2
2<3?  | 7 4 8 9 0 6 2
3<4?  | 7 4 8 9 0 6 2
4<5?  | 7 4 8 0 9 6 2
5<6?  | 7 4 8 0 6 9 2
      | 7 4 8 0 6 2 9  <-- After the first sweep.
                           The last entry of the array
                           is correct now.
```

BUBBLE: One Bubble Sweep, in C++

Compare neighbors, and swap if necessary.

```
for ( i = 0; i < n - 1; i++ )
{
  if ( list[i] > list[i+1] )
  {
      t          = list[i];
      list[i]    = list[i+1];
      list[i+1] = t;
  }
}
```

BUBBLE: Bubble Sort, One Sweep

If we carry out another sweep, you should believe me that the value 8, which is the second largest entry, ends up in its correct spot, second to last. Another sweep will put 7 in the correct place, then 6, and so on.

How many sweeps do we need to sort **n** items? We only need **n-1** sweeps, because when **n-1** numbers are in the right place, so is the last remaining one.

Moreover, each sweep is a little shorter than the previous one. Since the first sweep puts the correct number in the last place, the second sweep doesn't have to check there. The third sweep doesn't have to check the last two spots, and so on.

BUBBLE: Bubble Sort, One Step

Putting these ideas together, we have a code for the bubble sort:

```
int main ( )
{
  int a[100], i, last, n = 100, t;

  We assume the entries of a have been set.

  for ( last = n - 1; 0 < last; last-- )     <-- last points to the last unsorted element.
  {
    for ( i = 0; i < last; i++ )             <-- Compare neighbors up to last.
    {
      if ( a[i] > a[i+1] )                   <-- Is next element smaller than this one?
      {
          t      = a[i];                     <-- If so, swap them.
          a[i]   = a[i+1];
          a[i+1] = t;
      }
    }
  }
  return 0;
}
```

The **i** loop sweeps from 0 up to (but not including) **last**.
The **last** loop sweeps in decreasing lengths from **n-1** down to 1.

BUBBLE: Let's Only Write This Code Once!

Even though bubble sort is a simple idea, writing the code required some thought. The next time I need to sort an array, I will have to think through these ideas again.

But the code we've written now can be used to sort other arrays, so can't we avoid thinking and coding again?

We can get the algorithm written correctly once as a function that we can call whenever we need it.

Functions enable us to take a thought *(sort this array!)* and turn it into a C++ function that hides all the details. So when we write a program, and we need an array sorted, we realize we can just jot down *(sort this array!)* in our plans, meaning that the C++ c will call our sorting function.

Sorting, Polynomials

- Introduction
- Bubble Sorting
- **Insertion Sorting**
- Sending an Array to a Function
- Polynomials
- Assignment #6

INSERT: Insertion Sort

Now let us suppose that, although we know we are going to have **n** objects in our list, we start out with none at all and then receive our items one at a time.

This is just the way a hand of cards is dealt out, and you see impatient players taking each card immediately, and sorting it into proper order. This is an example of what is called insertion sort.

INSERT: Adding One Card at a Time

If we have just one card in our hand, there's nothing to do.

So things only get interesting when we have one card in our hand, and we are about to pick up a second card.

When the second card arrives, it might be

- smaller, so it goes to the left;
- larger, so it goes to the right;

Our hand is still sorted. When the third card arrives, it might be:

- smaller than both old cards, so it goes to the left;
- bigger than first, smaller than second, so goes in the middle
- larger than both, so it goes to the right;

INSERT: Thinking About an Algorithm

Because our hand is always sorted, adding a new card becomes easy. We simply have to determine where to insert it.

To think about this in terms of arrays, we first have to assume that the array is big enough to hold all the cards (or data items, really) that we expect to get, but that currently, we are "holding" just **n** values, in indexes 0 through **n-1**.

The new card will be assigned to array entry **in** (whatever that is), meaning that the cards in entries 0 through **in-1** don't move, but that all the cards currently in entries **in** through **n-1** have to shift one position to the right to make room for the new card.

INSERT: Algorithm Steps

Let's assume our sorted array can hold at most **n_max** entries, and that we currently have stored **n** entries.

Now assume we are given one new item to add to our array.

Inserting a single element into a sorted array involves steps:

- make sure **n+1** $\leq$ **n_max**;
- determine **in**, the index for the new item;
- shift elements **in** through **n-1** to the right;
- insert the new item at index **in**;
- update **n** to **n+1**.

INSERT: Determine the Insertion Index

Let's code the determination of **in**, the index for the new item.

We'll assume the array is called **a**, and the new item is **b**.

```
  in = n;                          <-- Let's guess that b is inserted at the end.

  for ( i = 0; i < n; i++ )        <-- To check, let's compare b against each array entry.
  {
    if ( b < a[i] )                <-- The first array entry that is bigger than b
    {                                  is in the spot where b should go
      in = i;
      break;
    }
  }
  return 0;
}
```

If **b** is bigger than all elements in **a**, it goes at the end, in index **n**.

Otherwise, if we start with the first element of **a**, as soon as we've found an element bigger than **b**, that is where **b** should go.

INSERT: Shifting Elements

Let's code the shift of elements **in** through **n-1**. Notice that there is a right and wrong way to do this!

If we start shifting from the lower end, the first value overwrites everything. But if we move the last entry to position **n**, that frees up a spot for the previous entry, all the way down to **in**.

```
for ( i = n - 1; in <= i; i-- )     <-- Starting at the end, move the item at index i to i+1.
{
  a[i+1] = a[i];
}

a[in] = b;                          <-- Now location in is open for b.
n = n + 1;                          <-- The number of items in the array went up by 1.
```

INSERT: An Example Code

insert_sort.cpp:

```
int a[100], b, i, in, n = 0, n_max = 100;

while ( n < n_max )
{
  cin >> b;                          <-- User enters one item at a time until CTRL-D.
  if ( cin.eof ( ) )
  {
    break;
  }
  in = n;                            <-- Find insertion point.
  for ( i = 0; i < n; i++ )
  {
    if ( b < a[i] )
    {
      in = i;
      break;
    }
  }
  for ( i = n - 1; in <= i; i-- )    <-- Shift data and insert.
  {
    a[i+1] = a[i];
  }
  a[in] = b;
  n = n + 1;                         <-- Update data count.

  for ( i = 0; i < n; i++ )          <-- Print current array.
  {
    cout << a[i] << "\n";
  }
}
```

INSERT: An Example Code

We've assumed the data is available one item at a time.

Could we also use insertion sort if we already had all the data?

Yes, with a few adjustments. Let **n_max** indicate the size of the full array. Let **n** indicate how many items of the array have been sorted so far, with **n** starting at because item 0 is sorted. Copy item 1 into the value **b**. That frees up entry 1 in the array. Now we can "insert" **b** into its correct position, either index 0 or index 1.

Now we have items 0 and 1 sorted. We copy item 2 into **b**, and insert it into its correct order with items 0 and 1.

In this way, we can insert the data one at a time, even though we have it all to begin with.

Sorting, Polynomials

- Introduction
- Bubble Sorting
- Insertion Sorting
- **Sending an Array to a Function**
- Polynomials
- Assignment #6

FUNCTIONS: Do It Right, But Only Once

It is convenient to have an array sorted. And sorting is another activity that is worth thinking about once, but not twice! When there's an operation that is useful, but somewhat tricky to put together, we should think about trying to write a function. That way, once we get the function written correctly, we never have to think about the underlying details.

Can we write a function that will sort an array for us? What is different about using a function when arrays are involved?

As we investigate this question, we will notice what seems like a significant difference in the way a function deals with input th represents an array.

FUNCTIONS: What Should the Interface Look Like?

When we write a function, the only part we will deal with later is the interface, or "header", that is, the information that describes the type and name of the function, and its input arguments.

It seems reasonable that the name of the function could be "bubblesort", and that it might need two input arguments: the dimension of the array, and the name of the array.

It's not clear what the output should be. You might think the output should be the sorted version of the array. This makes sense, given what we have seen in simpler examples, but because arrays are involved, it is not what we want to do. So instead, trust me that it will make sense if we leave the output as **void**.

```
void bubblesort ( int n, int a[] )
```

FUNCTIONS: Array Input Is Treated Differently

Notice how the array appears in the interface with square brackets.

That is how we tell C++ that this argument is an array. As we will see later, this is also the information that causes C++ to treat the array differently from the other input.

```
void bubblesort ( int n, int a[] )
```

Essentially, most input variables are only disposable copies, but when an array is input, it's as though C++ sends "the original copy", and the function can modify it, and those changes remain even after the function is done.

FUNCTIONS: The Function Code

bubblesort.cpp: Here is the code for the bubblesort function:

```
void bubblesort ( int n, int a[] )
{
  int i, last, t;

  for ( last = n - 1; 0 < last; last-- )     <-- last points to the last unsorted element.
  {
    for ( i = 0; i < last; i++ )             <-- Compare neighbors up to last.
    {
      if ( a[i] > a[i+1] )                   <-- Is element i+1 smaller than element i?
      {
          t      = a[i];                     <-- If so, swap them.
          a[i]   = a[i+1];
          a[i+1] = t;
      }
    }
  }
  return;                          <-- Void functions return nothing
}
```

FUNCTIONS: Using the Bubblesort Function

bubblesort_test.cpp: Here is a code to test bubblesort:

```
beginning stuff
int main ( )
{
  int a[10], i, n = 10;
  int random_int ( int a, int b );      <-- Must declare this function;
  void bubblesort ( int n, int a[] );   <-- Must declare this function;

  cout << "\nBefore sorting:\n\n";
  for ( i = 0; i < 10; i++ )
  {
    a[i] = random_int ( 0, 100 );
    cout << "  a[" << i << "] = " << a[i] << "\n";
  }

  bubblesort ( n, a );                  <-- a goes in unsorted, comes out sorted.

  cout << "\nAfter sorting:\n\n";
  for ( i = 0; i < 10; i++ )
  {
    cout << "  a[" << i << "] = " << a[i] << "\n";
  }
  return 0;
}
```

FUNCTIONS: Function Arguments

Up to now, we've talked about functions as purely input/output machines. The input comes in, an output value comes out. It seemed fairly simple. In particular, the only changes that a function could make would come from its output value.

What is peculiar about **bubblesort** is that I passed the array **a[]** as input, and I expect that the function will rearrange the data and return it to me. Is that what happens *(in this case, yes)*

Can a function change any of its input arguments? *(yes, locally.)*

Will this change affect the values of those items in the calling program? *(usually not!)*

FUNCTIONS: Function Arguments

In C++, when a function has input arguments that are scalars (things that are not arrays), the function simply receives a separate copy of the information. This copy is in the memory space of the function, and disappears on return from the function. Thus, the function can change those values, but the changes have no effect "in the outside world", that is, in the calling program.

However, when a function has an array as an input argument, C++ does not make a copy of the array. Instead, what happens is that C++ sends the function a copy of the "address" of the array, that is, how to find the values. The array itself is not copied, and the function can modify those values directly.

Thus, scalar input arguments are strictly input. But array input arguments can be changed by the function in a way that affects their value when the function is completed.

FUNCTIONS: Example of Memory Usage

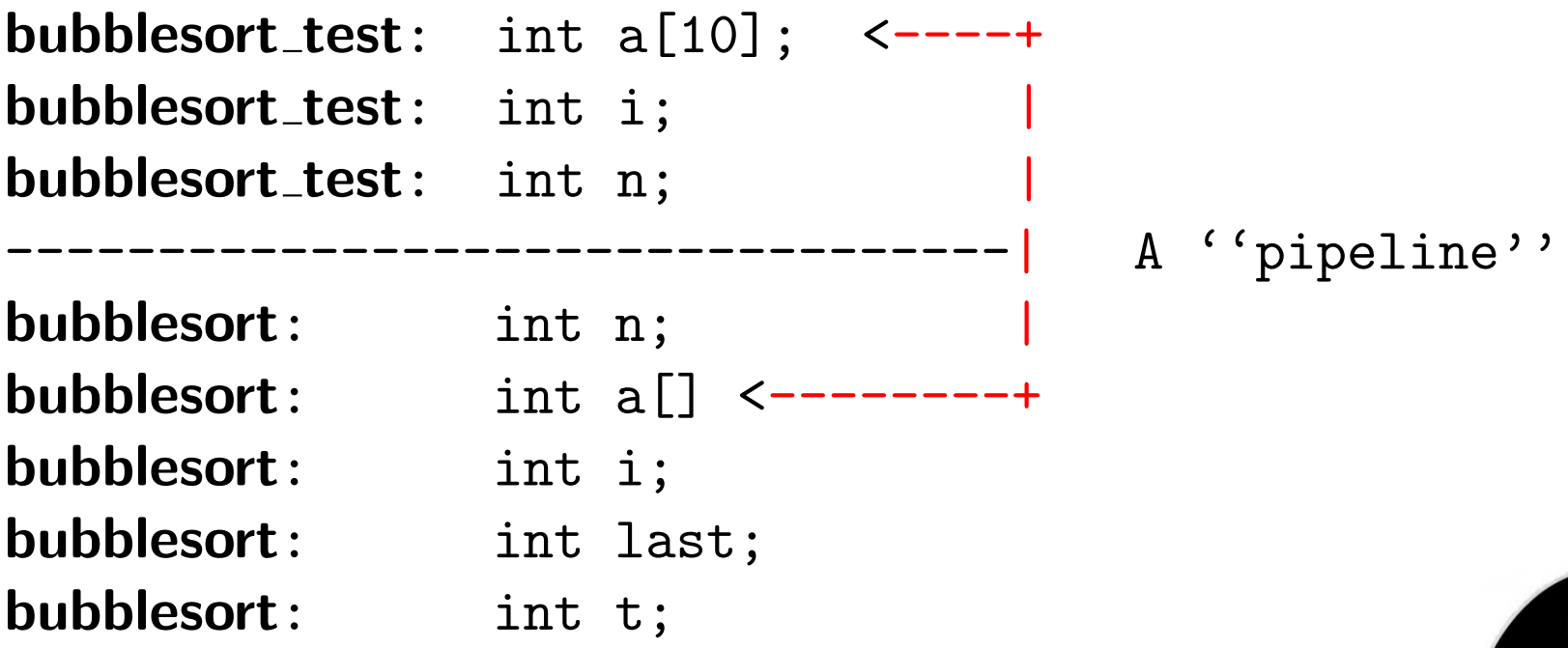
To see what this might look like, let's consider how C++ thinks of the variables in the **bubblesort_test** and **bubblesort** functions.

When **bubblesort_test** begins, C++ creates variables

bubblesort_test: `int a[10];` `filled with data`
bubblesort_test: `int i;` `used by the` **for** `loop.`
bubblesort_test: `int n;` `set to 10.`

FUNCTIONS: Example of Memory Usage

When **bubblesort_test** calls **bubblesort(n,a)**, C++ creates variables as follows:



FUNCTIONS: Example of Memory Usage

There is only one array **a**. The names in **bubblesort_test** and **bubblesort** are connected by a sort of "pipeline". Any changes made by **bubblesort** to the **a** array will show up in the calling function **bubblesort_test**.

Both functions have a variable called **n**. But this is not an array. So **bubblesort** made a separate copy, which starts out equal to the value assigned on input. The **bubblesort** function could change the value of **n**, but these changes would only be local, and would vanish when all the **bubblesort** memory is wiped out.

So in the simplest case, you can assume that a C++ function can affect input arguments that are arrays, but not other kinds.

If you need a C++ function to modify input arguments and not them, there is a way to do this, known as a "call by reference". We will look at this later.

Sorting, Polynomials

- Introduction
- Bubble Sorting
- Insertion Sorting
- Sending an Array to a Function
- **Polynomials**
- Assignment #6

POLYNOMIALS:

In mathematics, a **polynomial** is a special kind of function of **x** which can be written as a linear combination of powers of **x**.

Examples:

- $p_1(x) = x^2 + 3x + 2$;
- $p_2(x) = x^8 - 256$;
- $p_3(x) = 3x - 7$;
- $p_4(x) = 5$.

The highest power of x that occurs in a polynomial is known as its degree, d. For the four examples, the value of the degree d is 2, 8, 1 and 0, respectively.

A polynomial of degree d can always be written in the form:

$$p(x) = c_0 + c_1 x + c_2 x^2 + ... + c_d x^d$$

POLYNOMIALS:

It is natural, in C++, to represent a polynomial of degree d by an array of dimension $d + 1$. Here is how we might declare arrays that store the information defining the example polynomials:

```
p_1: double c1[3] = { 2, 3, 1 };
p_2: double c2[9] = { -256, 0, 0, 0, 0, 0, 0, 0, 1 };
p_3: double c3[2] = { -7, 3 };
p_4: double c4[1] = { 5 };
```

POLYNOMIALS:

If you want to plot a polynomial, or make a table of its values, you need to use the coefficient array **c[]** to evaluate $p(x)$.

Here is one way:

```
double c[d+1];              <-- Replace "d+1" by a number.


p = 0.0;
xi = 1.0;
for ( i = 0; i <= d; i++ )
{
  p = p + c[i] * xi;
  xi = xi * x;
}
```

POLYNOMIALS: Derivatives

The derivative of a polynomial

$$p(x) = c_0 + c_1 x + c_2 x^2 + ... + c_d x^d$$

is

$$p'(x) = c_1 + 2 * c_2 x + 3 * c_3 * x^2 ... + d * c_d x^{d-1}$$

If we are interested in the derivative, we could set up a coefficient array for it, of dimension d, as follows:

```
double cp[d];                    <-- Replace "d" by a number.
for ( i = 0; i < d; i++ )
{
  cp[i] = ( i + 1 ) * c[i+1];
}
```

POLYNOMIALS: Product of Polynomials

Suppose $p_1(x)$ and $p_2(x)$ are polynomials of degrees d_1 and d_2. If we compute the product $p(x) = p_1(x) * p_2(x)$, we get a polynomial of degree $d = d_1 + d_2$. The value of the coefficient **c[i]** can be found by making a table in which we pair coefficients **c1[j]** and **c2[j]** for which $i = j + k$.

Suppose we multiply $p_1(x) = x^2 + 3 + 2$ times $p_2(x) = 3x - 7$. The table suggests the terms we are going to get.

```
          2       3x        1x^2
      ---------------------------
-7   | -14     -21x      -7x^2
 3x  |   6x     9x^2     3x^3
```

For a polynomial with more coefficients, this might seem to get complicated!

POLYNOMIALS: Product of Polynomials

It is actually a simple matter to automatically compute these terms and group them together correctly:

```
double c[d1+d2+1];                  <-- Replace d1+d2+1 by a number;

for ( i = 0; i <= d; i++ )
{
  c[i] = 0.0;                       <-- Zero out the new array.
}

for ( i1 = 0; i1 <= d1; i1++ )      <-- Pick a coefficient of p1(x)
{
  for ( i2 = 0; i2 <= d2; i2++ )    <-- Pick a coefficient of p2(x)
  {
      i = i1 + i2;                  <-- What is the power of x?
      c[i] = c[i] + c[i1] * c[i2]; <-- Add to the coefficient of that power.
  }
}
```

For some later problems, we will need these polynomial results!

Sorting, Polynomials

- Introduction
- Bubble Sorting
- Insertion Sorting
- Sending an Array to a Function
- Polynomials
- **Assignment #6**

ASSIGNMENT #6: Football Game Attendance

The file **fsu_football_2010.txt** contains the attendance records for 13 FSU football games in the 2010 season.

Write a program which reads each attendance value from the file, stores the attendance values into an array, and:

- **prints** each attendance value as it is read;
- **sorts** the attendance values, and prints the sorted list;
- **averages** the attendance values and prints the average.

ASSIGNMENT #6: Requirements

Your program *must* read its input from the text file. Your program should assume that up to 20 values might be read; it should determine how many values there actually are in the process of reading the data.

Your program must use the insertion sort method to sort the data. As soon as one data item is read, it should be printed, then sorted.

The output from your program must be in three **labeled** groups:

1. "Attendance Values" lists attendance values in the order read;
2. "Sorted Attendance Values" lists attendance values in sorted (increasing) order;
3. "Average Attendance" should print the average attendance

ASSIGNMENT #6: Things to Turn in

Email to Detelina:

- your program's results;
- a copy of your program.

The program and output are due by Thursday, July 7.