Strings, 2D Arrays, Matrices, Images

http://people.sc.fsu.edu/~jburkardt/isc/week08_lecture_15.pdf

.....

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

.....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 28 June 2011

Strings, 2D Arrays, Matrices, Images

- **Introduction**
- Strings
- 2D Arrays in C++
- Matrix Operations
- Image Files
- Discussion of Midterm

INTRO: Schedule

Read:

- Today's class covers Sections 6.8, 6.11

Next Class:

- Solving a Nonlinear Equation

Assignment:

- Thursday, July 7: Programming Assignment #6 is due.

Midterm Exam:

- Thursday, June 30th

INTRO: Strings, 2D Arrays

We will start today by talking about C++ strings, which provide a data type that is convenient for working with text. We have already seen the very simple **char** data type, which works with one character at a time, but the **string** type is more powerful. It turns out that a string is a kind of array, that is, a list of characters.

We will also look at 2D arrays, which allow us to handle information that is best described by a pair of indices, perhaps the row and column values. A common example occurs when information is arranged on a grid, which might be a checkerboard, a map, a list of student grades for a series of tests, and so on.

INTRO: Matrices, Images

In mathematics and physics, what C++ calls 1D and 2D arrays are instead termed *vectors* and *matrices*. Matrices are used to define systems of linear equations, or to specify a stress field, or some other operation. Matrices transform vectors to new vectors by matrix-vector multiplication.

We have already seen several examples of simple graphics images, in which a grid is imposed on a picture, and a color, gray scale, or black/white value is recorded for each grid cell. The resulting information is best saved as a 2D array. We will look at how a C++ program might work with such information.

Finally, we'll talk about the midterm exam coming on Thursd

Strings, 2D Arrays, Matrices, Images

- Introduction
- **Strings**
- 2D Arrays in C++
- Matrix Operations
- Image Files
- Discussion of Midterm

STRINGS: Recall the CHAR Data Type

We have already seen a data type called **char** which C++ uses to store single characters. We think of a character as the letters of the alphabet, but C++ includes numbers, punctuation, spaces, and even the carriage control character.

Recall that a literal character is marked off by single quotes:

```
char alpha = 'a', beta = 'b', gamma = '@';
```

and that we read characters from the Gettysburg Address file using a familiar command like:

```
cin >> alpha;
```

but to include blanks, new lines and everything, we had to use

```
alpha = cin.get ( );
```

STRINGS: Strings Are Like Words

When working with text, we think of words, not characters. So it's important that C++ includes a data type that can handle words or, in general, strings of text.

Some facts about strings:

- a string is made up of characters, in definite order so we can think of it as a kind of array;
- we may not be sure how long a string should be;
- the string may get longer or shorter as we work on it;
- we may want to insert or change characters in the string;
- we want to be able to alphabetize strings.

STRINGS: Longest Word in a File

Let's look at how C++ allows us to work with strings by repeating a the experiment we did earlier, where we searched the Gettysburg address for the longest word. That time, we only knew about characters, and so we had to read one character at a time, and we knew a word had ended the character we read was not alphabetic (and so probably a space or a carriage return.)

This time, we can define a variable **w** that is a C++ string. Then when we ask **cin** to read a value for **w** from the file, it will read an entire word (or what it thinks is a word!) in one operation.

C++ has a simple command to report the length of a string, so we can find out how long the word **w** is. If we keep track of the longest value so far, our program is done.

STRINGS: Find Longest Word Using Strings

longest_word2.cpp:

```
# include <cstdlib>
# include <iostream>
# include <string>                    <-- Required when using strings.

using namespace std;

int main ( )
{
  int length, int length_max = 0; <-- Initialize longest word length to 0.
  string w, w_max = "";           <-- Initialize longest word to blank.

  while ( true )
  {
    cin >> w;                     <-- Read another word from the file.

    if ( cin.eof ( ) )
    {
      break;
    }
    length = w.length ( );        <-- How C++ measures string length.

    if ( length_max < length )
    {
      length_max = length;
      w_max = w;
    }
  }

  cout << "Longest word is " << w_max << "" which has length " << length_max << ".\n";
  return 0;
}
```

STRINGS: Comments on the Program

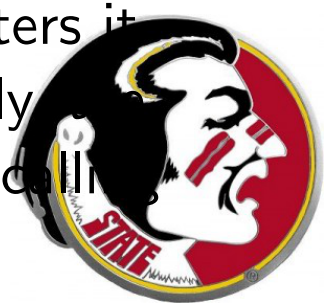
A program that uses C++ strings must have have the statement

```
# include <string>
```

We mentioned that a literal value of type **char** is indicated by using single quotes around a character.

For strings, a literal value is indicated using double quotes. In other words, all the labels we have used in our **cout** statements, all the way back to **"Hello, world!"**, have been strings. In the program, the only string literal we see is when we initialize **w_max** to the empty string "".

The length of the string variable **w** is the number of characters it currently contains, and is computed by **w.length()**. Similarly, the length of a string variable called **fred** would be returned by calling **fred.length()**.



STRINGS: Is a String Empty?

There are many functions available for operating on strings, and most of them are used in a similar way. For instance, there is a function called **empty()** which is true if a string is empty. To find out if the string **barney** is empty, we group the name of the string, a period, and the name of the function together:

```
if ( barney.empty ( ) )
{
  barney = "I was empty!";
}
```

STRINGS: Extract a Substring

The **substr()** function copies part of a string:

- **word.substr(start)** copies **word** from location **start**;
- **word.substr(start,n)** copies **n** characters from **word**, beginning at location **start**.

The characters in a string are like elements in an array. They are indexed starting with 0, and we can retrieve the value of any element using index notation: **s[5]** is the 6th character in string **s**.

We can "add" strings using the "+" sign.

```
s1 = "Tallahassee";
s2 = s1.substr ( 1 );        <-- s2 is now "allahassee"
s3 = s1.substr ( 5, 4 );     <-- s3 is now "hass";
s4 = s2 + s1[0] + "ay";      <-- s4 is now "allahassee" + "T" + "ay"
                                 = "allahaaseeTay"
```

STRINGS: The Piggysburg Address

From what I have told you, you should know enough about C++ strings to be able to make a Pig Latin version of the Gettysburg Address.

Remember that to turn a word into Pig Latin:

- If the word begins with a consonant sound, the consonant sound is moved to the end, and followed by "ay";
- If the word begins with a vowel sound, the word has "way" added to the end.

Once you have done this, your text *illway artstay otay ooklay idiculousray*.

Luckily, a computer is not too proud to do such operations.

Strings, 2D Arrays, Matrices, Images

- Introduction
- Strings
- **2D Arrays in C++**
- Matrix Operations
- Image Files
- Discussion of Midterm

2D: A 1D Array is a List with 1 Index

The idea of a list allowed us to create a single variable to store many values, while giving us an *index* to access any particular value we wanted.

It's important that there be a simple relationship between the location of the desired value and the computation of the index. We have seen many examples in C++ now, in which a list of, say, 1000 numbers was stored in such a way that the 10th item had index 9, the 100th item had index 99 and so on.

2D: A 2D Arrays Uses Two Indices

There are many kinds of data which are naturally two dimensional, so that it is much more logical to be able to index the data by specifying two values rather than one. As an example, think of the file containing the image of the Mona Lisa as a table of gray scale values, organized into 250 columns and 360 rows. That makes 90,000 pixels. But would I rather ask for pixel 45,000 or for the pixel in row 30, column 1?

I would like C++ to be able to store 2D data in a way that allows me to access it by row and column indexes.

Such a quantity is called a *two dimensional array*, and its use is a simple generalization of what we already know about what we might call "one dimensional arrays".

2D: An Example of a 2D Array

A 2D array is a C++ variable which has a name, a type, and a pair of dimensions, which we might symbolize as **m** and **n**. The first dimension is known as the *row* index, and the second as the *column* index. Often, the names *i* and *j* are used for typical row and column indexes.

A 2D array can be thought of as a table. Here is an example of a 5x4 array, which we will call **sam**:

```
            columns
       | j: 0   1   2   3
     i +----------------
r    0 |    0   1   2   3
o    1 |   10  11  12  13
w    2 |   20  21  22  23
s    3 |   30  31  32  33
     4 |   40  41  42  43
```

2D: Declaring and Initializing a 2D Array

The **sam** array can store **m** * **n** = 5x4 = 20 entries.

The declaration for the array would be

```
int sam[5][4];
```

A row index for **sam** is any value from 0 to 4; column indexes run from 0 to 3. **sam[4][2]** is legal, but **sam[2][4]** is not!

Initializing a 2D array is more complicated than the 1D case. You need to give a list, in curly brackets, and inside this list, you give the values for each row, again in curly brackets.

Thus, the **sam** array could be initialized by:

```
int sam[5][4] = { {  0,  1,  2,  3 },
                  { 10, 11, 12, 13 },
                  { 20, 21, 22, 23 },
                  { 30, 31, 32, 33 },
                  { 40, 41, 42, 43 } };
```

2D: Accessing entries of a 2D Array

Since the entries of a 2D array have two indexes, one way to set or read or print them all is to use a double **for** loop, with an **i** loop on the rows and a **j** loop on the columns.

It turns out that the entries of **sam** can be computed this way:

```
m = 5;
n = 4;

for ( i = 0; i < m; i++ )
{
  for ( j = 0; j < n; j++ )
  {
    sam[i][j] = 10 * i + j;
  }
}
```

Thus, sam[4][1] = 10 * 4 + 1 = 41;

2D: Reading a 2D Array

Just as with a 1D array, the **cin** >> operator can read the values of a 2D array from a user or a file. We do this one entry at a time, using a double **for** loop, typically using indices called **i** and **j**:

```
int ann[3][5];

cout << "Enter 3 rows of 5 numbers:";
for ( i = 0; i < 3; i++ )
{
  for ( j = 0; j < 5; j++ )
  {
    cin >> ann[i][j];
  }
}
```

We read the 5 entries in row 0: [0][0] through [0][4], and then row 1, then row 2.

2D: Printing a 2D Array

It's natural, for a small array, to try to print each row on one line:

```
int sam[5][4];

for ( i = 0; i < 5; i++ )
{
  for ( j = 0; j < 4; j++ )
  {
    cout << "  " << sam[i][j];
  }
  cout << "\n";
}
```

2D: Using a 2D Array

Any element of a 2D array can be used in arithmetic computations, logical checks, and assignment statements just like any other variable.

```
for ( i = 0; i < student_num; i++ )
{
  for ( j = 0; j < test_num; j++ )
  {
    if ( 100.0 < grade[i][j] )
    {
      grade[i][j] = 100.0;
    }
    else if ( grade[i][j] < 0.0 )
    {
      grade[i][j] = 0.0;
    }
  }
}
```

2D: Recall 1D Arrays as Function Arguments

Passing a 2D array to a function is a little trickier than was the case for a 1D array.

For the 1D case, the function call involved simply placing the array name in the argument list:

```
bubblesort ( n, a )
```

The function declaration was only slightly strange. For an array input, we had to give a name and a pair of empty square brackets.

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

2D: 2D Arrays as Function Arguments

For the 2D case, suppose function **array_sum()** should add up all the elements in an array of M rows and N columns. We would expect to use such a function this way:

```
total = array_sum ( m, n, b );
```

To declare the function, you might expect simply:

```
void array_sum ( int m, int n, int b[][] ); ??? (No!)
```

However, for technical reasons, the second pair of square brackets cannot be empty; it must contain the actual number of columns in the array, and as a number, not a symbolic variable.

```
void array_sum ( int m, int n, int b[][10] );
```

And that means your function can only be used for arrays with exactly 10 columns.

2D: 2D Arrays of Fixed Size as Function Arguments

We will be interested in problems involving 2D arrays, but if we want to use a function to print the array or analyze it or update it, we will have to decide in advance exactly how big the array will be.

That's not too bad. We might be working on an 8x8 checkerboard, or a 20x20 minesweeper field *(these are possible project areas, by the way!)*. As long as the size of the array is fixed, we can write and use functions in a simple way.

To be able to deal with 2D information in a smoother way, you will need to wait for your next C++ course!

Strings, 2D Arrays, Matrices, Images

- Introduction
- Strings
- 2D Arrays in C++
- **Matrix Operations**
- Image Files
- Discussion of Midterm

MATRIX: Mathematical Vectors and Matrices

In mathematics, physics and other sciences, we have the concepts of *vector* and *matrix*. As far as we're concerned, a vector is simply a list of numbers of a fixed length, and a matrix is table with given number of rows and columns. In other words, they're both simply 1D and 2D arrays.

Let's just consider a few common operations involving vectors and matrices, to show how they are implemented in a programming language.

MATRIX: Euclidean Length

The **norm** or **Euclidean length** of a vector is the square root of the sum of the entries.

```
double vector_norm ( int n, double a[] )
{
  int i;
  double value = 0.0;

  for ( i = 0; i < n; i++ )
  {
    value = value + a[i] * a[i];
  }
  value = sqrt ( value );
  return value;
}
```

MATRIX: Dot Product

The **dot product** or **inner product** of two vectors is the sum of the products of corresponding entries:

```
double vector_dot ( int n, double a[], double b[] )
{
  int i;
  double value = 0.0;

  for ( i = 0; i < n; i++ )
  {
    value = value + a[i] * b[i];
  }
  return value;
}
```

MATRIX: Matrix Vector Product

A **matrix vector product** of the form **A*x=b** multiplies an **m**x**n** matrix **A** by an **n** vector **x**, and computes an **m** vector **b**.

This example can't easily be made more general, since a 2D array is involved.

```
double a[10][5], x[5], b[10];
int i, int j, m = 10, n = 5;
```

Here, we assume, **a** *and* **x** *get assigned values.*
Now we can multiply them to compute **b**.

```
for ( i = 0; i < m; i++ )
{
  b[i] = 0.0;
  for ( j = 0; j < n; j++ )
  {
    b[i] = b[i] + a[i][j] * x[j];
  }
}
```

MATRIX: Solving A*x=b for x

Suppose we computed a matrix vector product of the form **A*x=b**, and now, somehow, we've lost the value of **x**, but we still know **A** and **b**. Can we figure out what **x** was? Sometimes we can, especially if the matrix is square. The tool for doing this is known as **Gauss elimination**, and involves combining rows of the matrix to zero out entries.

You may have seen this topic in high school algebra, or in a mathematics class here.

It's an interesting problem to try to set up Gauss elimination in C++. It requires some careful planning and thinking.

This is another topic that would be suitable for a final projec

Strings, 2D Arrays, Matrices, Images

- Introduction
- Strings
- 2D Arrays in C++
- Matrix Operations
- **Image Files**
- Discussion of Midterm

IMAGES: Images Are 2D Arrays

A common example of a 2D array occurs with image files:

- a **bit map** image, in which each entry is 0 or 1, and the image contains pixels that are black or white,
- a **gray scale** image, in which each entry is between 0 and 255, with low numbers dark and high numbers light.

In our class exercise, we read a gray scale image, but we did not know how to save it as a 2D array. Let us work through those details now, and see how we can transform the image with simple mathematical tricks!

IMAGES: The Mona PGM File

The file **mona.pgm** contains a portable gray map representation of the Mona Lisa, broken down into 250 columns and 360 rows of integers between 0 and 255.

If we wanted to read this data, we need to create an array with the appropriate name (mona), type (int), and 2 dimensions (360 rows by 250 columns):

```
int mona[360][250];
```

IMAGES: Reading the Mona PGM File

Since the data in the file is stored by rows, that means that as we read the data, the first 250 numbers we get go into row 0, and then we move on to the next row. So we read with a double **for** loop, and the row loop is on the outside, and the column loop on the inside:

```
for ( i = 0; i < 360; i++ )
{
  for ( j = 0; j < 250; j++ )
  {
    cin >> mona[i][j];
  }
}
```

IMAGES: Reversing the Mona Data

Let's make a "reverse video" copy of **mona** in which black and white are switched. If a pixel of the array has the value **g**, then we want our copy to have the value 255-**g**.

After we read **mona**, we reverse the data;

```
for ( i = 0; i < 360; i++ )
{
  for ( j = 0; j < 250; j++ )
  {
    mona[i][j] = 255 - mona[i][j];
  }
}
```

IMAGES: Write Out the Reverse Image

To write the data out, we have to write three lines of information first, containing the code **P2**, then the number of columns and rows, then the maximum gray scale value:

```
cout << "P2" << "\n";
cout << 250 << "  " << 360 << "\n";
cout << 255 << "\n";

for ( i = 0; i < rows; i++ )
{
  for ( j = 0; j < cols; j++ )
  {
    cout << " " << mona[i][j];
  }
  cout << "\n";
}
```

IMAGES: The Reverse Image

mona_reverse.cpp:

IMAGES: Frame the Face

I hope the images make it a little easier to think about working with 2D arrays.

As a second experiment, let's try to put a white frame around the face. There are 360 rows. I'd say the head is between rows 30 and 120. There are 250 columns. I'd say the head is between columns 75 and 175.

Let's make a crude frame by setting these rows and columns white.

To "white out" all the pixels in row 30, we need a single **for** loop that runs across the row **i** = 30, with **j** going from 0 to 250.

IMAGES: Frame the Face

```
//
//  White out rows 30 and 120, columns 75 and 175.
//
  for ( j = 0; j < cols; j++ )    <-- Every pixel in this row.
  {
    mona[30][j] = 255;
    mona[120][j] = 255;
  }

  for ( i = 0; i < rows; i++ )    <-- Every pixel in this column.
  {
    mona[i][75] = 255;
    mona[i][175] = 255;
  }
```

IMAGES: The Framed Image

mona_frame.cpp:

IMAGES: Discussion

Gray scale images are easy to work with as 2D arrays, particularly if you can first convert them to the PGM format. (A program like ImageMagick **convert** will do this, for instance.)

There are many algorithms for improving, analyzing, and transforming images. They are important because images store medical data, spy satellite reports, agricultural surveys, particle physics collisions.

But images are often just data, not information, until a program can analyze the data and find meaning in it.

Image analysis is another suitable topic for the final project.

Strings, 2D Arrays, Matrices, Images

- Introduction
- Strings
- 2D Arrays in C++
- Matrix Operations
- Image Files
- **Discussion of Midterm**

MIDTERM: Rules

Here are the rules for the midterm exam:

1. You may bring two sheets of paper that contain notes. These two sheets of paper must have your name on them. You may refer to these two pages of notes during the exam;
2. No other reference items are allowed.
3. There will be 11 questions; answer 10 of them, and cross out the number of the one you want to skip.
4. I will not ask you to write an entire program; but I will ask for loops, functions, or groups of statements that do something. Include all the correct punctuation, variable declarations, and initializations that would be added to a complete program.
5. If I ask you to write a short function, I want everything: th header (first line), the curly brackets, declarations, calculations, and the return of the value.

MIDTERM: Coverage

Here are the topics I would suggest you review for the exam:

1. know your **for** loops backwards and forwards!
2. a **while** loop can loop while a condition is true, or can run "forever" but **break** out;
3. use **if/else** statements to control a calculation;
4. read data from a user;
5. print a list or array of data, perhaps with labels, or in groups;
6. find the index in an array where something is true;
7. count the elements of an array for which something is true;
8. know rules for defining a simple function;
9. know how to compute elements of a sequence;
10. know how we used random numbers to estimate and simul[illegible];

MIDTERM: Less Important

Here are the topics you should worry less about:

1. **char** and **bool** variables;
2. **switch** statement;
3. CTRL-D and end-of-file conditions
4. sorting;
5. graphics;
6. # include files;
7. math functions;
8. order of operations in arithmetic
9. Material covered today will not be on the exam at all;