Chapter 8

Arrays

Calculate the Mean and Standard Deviation for an Exam

Mean (average)

$$\text{Mean} = \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Standard Deviation (distance from the mean)

$$\text{Std. Dev.} = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n}}$$

where

x_i represents the individual exam scores

$\bar{x}$ represents the mean

In previous examples where an average was desired, we needed only an accumulator to add up the scores so an average could be calculated. We had no need to keep the individual scores for later use. If the standard deviation is to be calculated we need not only the average but also each individual score.

One-Dimensional Array
(a composite data type)

Thus far all of the variables used in our programs have been ***simple*** data types. A simple variable may store only one value at a time under its name. For a variable to store multiple values under one name it must be of a ***structured*/composite** data type. The structured type, the array will do well for the statistics example.

Array - A contiguous, homogeneous collection of data values that share a common name. An array is often referred to as a subscripted variable. Elements stored in an array are accessed by their name followed by the position number (subscript) in the array. The first element in the array is called the zeroth element.

The statement

```
cout << ages[0];
```

would output the first age contained in the array called ages.

The position number (contained in square brackets) is called the subscript and represents the number of elements from the beginning of the array. So, the second element in the array would be referenced as ages[1], one element from the front. A subscript must be an integer, or an integer expression using any integral type.

One-Dimensional Array - a structured/composite type that may store a list of values all of the same data type.

Declaring Arrays

DataType ArrayName[NamedOrLiteralIntConstant];

Given the following declaration:

float scores[25];

- the name of the array is scores
- each element is a float value
- there are 25 elements
- the subscripts range from 0 - 24

Accessing Elements in an Array

ArrayName[Sub-scriptExpression]

In the array declaration the size of the array had to be a named or literal constant of type int. When the array is accessed, the sub-script may be a named constant, a literal constant, an expression or a function call. Just remember that the end result must be an integer value.

```
scores[3] = 10;
averages[0] = 92.5;
cout << scores[3];
scores[3 + 2 - 1] = scores[6];
cin >> scores[i];
```

Initializing an Array

This example initializes the array by reading from the keyboard.

```
int nums[5];

for(int i = 0; i < 5; i++)
{
    cout << "Enter an integer: ";
    cin >> nums[i];
}
```

An array may be initialized in the declaration by following the declaration with an equal sign and a list of initializers (separated by commas) enclosed in French braces.

- If the number of initializers is less than the number of elements in the array, the remaining elements are initialized to zero. At least one initializer must be specified.

  ```
  int nums[5] = {5, 2, 4};
  ```

  nums contains the values 5, 2, 4, 0, and 0

- If the number of initializers is more than the number of elements, a syntax error results.

  ```
  int nums[5] = {5, 2, 7, 8, 3, 4};       // compiler error
  ```

- If no array size is specified, the size of the array is equal to the number of initializers. DON'T DO THIS.

  ```
  int nums[ ] = {4, 2, 7, 3, 9, 6};
  ```

Sample program to declare an array, load the array and print the contents.

```
#include <iostream.h>
void main(void)
{
    float gpa[5];            // an array holding 5 grade point averages - INPUT & OUTPUT

    // load the array from the keyboard
    for(int i = 0; i < 5; i++)
    {
        cout << "Enter the gpa for student " << i+1 << ": ";
        cin >> gpa[i];
    }

    // output the contents of the array
    cout << "\n\nStudent Grade Point Averages\n";
    for(int i = 0; i < 5; i++)
        cout << "\nGPA for student " << i+1 <<": " << gpa[i];
}
```

Array Exercise
Mean & Standard Deviation (p. 9-2)

1. Write the necessary declarations for a structure that will hold 10 quiz scores (the number of scores should be a named constant). The quiz scores are of type int. Declare also locations to hold the mean and standard deviation of these scores.

2. Draw the flowchart for the main function.

3. Write the function prototype for a function called InputScores that obtains the scores from the user, loads them into the array and returns the array to the calling function. Draw the flowchart for this function.

4. Write the function prototype for a function called CalcMean that calculates the mean of the scores and returns this value to the calling function. Draw the flowchart for this function.

5. Write the function prototype for a function called CalcStDev that calculates the standard deviation of the scores and returns this value to the calling function. Draw the flowchart for this function.

6. Write the function prototype for a function called OutputStats that outputs the scores, the mean of the scores and the standard deviation. Draw the flowchart for this function.

A Brief Introduction to the String Class

Think of a class as a template for potential data to be stored and the set of operations that will manipulate this data. We have looked at strings as an array of characters. This has worked fine thus far but things get a little more complicated when we get into creating arrays of strings and passing strings as parameters. To use the string class you must include <string.h> in your program. We will delay the discussion of classes until later in the course. For now, we are simply going to start using the string class in our programs.

Example:

```
#include <string>
using namespace std;
void main(void)
{
    string name;
}
```

Note that the number of characters is left unspecified and that the string is no longer an array of characters terminated with the null character. Note also that we need to use a different function (the getline function) to read strings that contain blank spaces.

```
getline(cin, name);
```

The first argument must be an input stream variable and the second argument a string variable. This function does NOT skip leading white space and continues to read until it encounters the newline character. **The newline character is removed from the buffer but is not stored as part of the string.**

Because leading white space is not ignored (recall this includes blanks, newline characters and tabs) it is often necessary to use the ignore( ) function when we use the string class. Recall

```
cin.ignore(100,'\n');
```

would ignore the next 100 characters OR until the newline character was encountered.

Example:

```
{
    string empName;
    int age;
    string socSecNum;

    cout << "Enter employee name: ";
    getline(cin, empName);
    cout << "Enter age: ";
    cin >> age;
    // get the newline out of the buffer before reading the ss#
    cin.ignore(1,'\n');
    getline(cin, socSecNum);
    etc.
}
```

Additionally, the use of the string class allows us to compare two strings without using the strcmp( ) function. Given

```
string name1;
string name2;
```

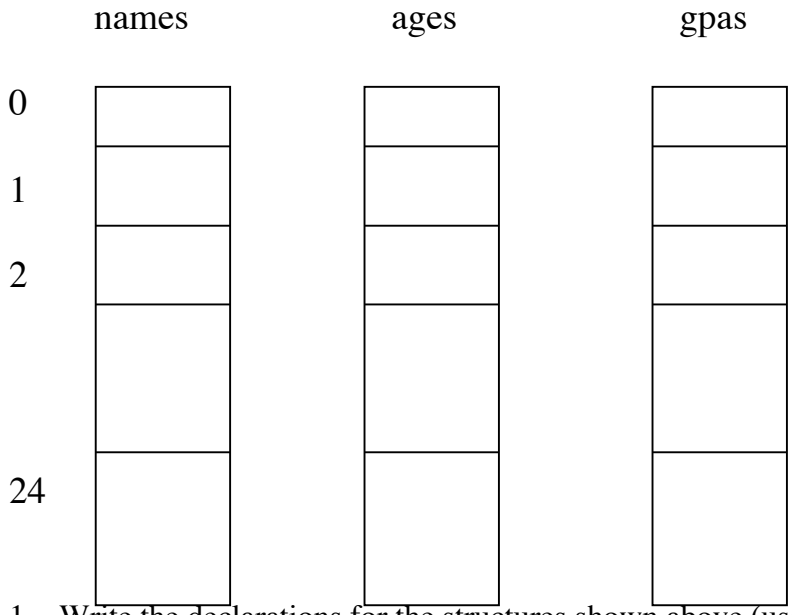
the statement

```
if (name1 == name2)
```

would be valid

Parallel Arrays

In CS 1A we worked with a database package. One way to implement a database is to use parallel arrays. Because the elements of an array must all be of the same type it is not possible to store names and ages for example in the same array. Given the following



1. Write the declarations for the structures shown above (use the string class for the array of names).

2. This program will contain a function to load the arrays, a function that searches the gpas array and returns the address of the highest gpa, and the main function. The main function will call the load function, the find highest gpa function and then output the name, age and gpa of the person with the highest gpa. Assume no two people have the same gpa.

3. Rewrite the load function so that it loads information about an unknown number of people into the arrays. You will need to count the number of people as their information is entered and you will not be able to use a for loop.

Aggregate Operations

Aggregate Operation - an operation on a data structure as a whole.

Aggregate operations on arrays are not allowed in C++. Given

```
int arrayOne[20];
int arrayTwo[20];
```

```
arrayOne = arrayTwo;                // assignment - not allowed

if(arrayOne == arrayTwo)            // comparison - not allowed

cout << arrayOne;                   // output - not allowed (except C-strings)

cin >> arrayOne;                    // input - not allowed (except C-strings)

arrayTwo = arrayTwo - arrayOne;     // arithmetic - not allowed

return arrayOne;                    // returning an entire array - not allowed

SomeFunc(arrayTwo);                 // pass as an argument to a function - allowed
```

Cstrings and Aggregate Operations

C++ allows some aggregate operations on char arrays called C-strings. A C-string is a null-terminated sequence of characters stored in an array of characters. The null-terminator ('\0' - digit 0) is used to mark the end of a string. The following counts the number of characters in a string, not including the null-terminator. Note that I/O is supported for C-strings.

```
{
    char myString[20];
    cout << "Enter a string: ";
    cin.getline(myString,20);
    int count=0;
    while(myString[count] != '\0')
        count++;
    cout << "\n\nThere are " << count << " characters in the string " << myString;
}
```

There are some C-string intrinsic functions located in cstring.h that may be used to copy strings, compare strings, find the length of a string etc.

strlen(str)	returns an integer representing the length of the string (null-terminator excluded)	
strcmp(str1,str2)	returns	an integer < 0 if str1 < str2 the integer 0 if str1 = str2 an integer > 0 if str1 > str2
strcpy(toStr, fromStr)	copies fromStr into toStr (null-terminator included), overwriting what was there - toStr must be large enough to hold fromStr	

```
void main(void)
{
    char strOne[5];
    char strTwo[8];
    cout << "Enter a string: ";
    cin.get(strOne,5);                    // seems to be more reliable than cin.getline( )
    cin.ignore(100,'\n');
    cout << "Enter a string: ";
    cin.get(strTwo,8);
    if(strcmp(strOne,strTwo)==0)
        cout << "Strings are the same\n";
    else
        if(strcmp(strOne,strTwo)<0)
            cout << "First string is less than second string\n";
        else
            cout << "First string is greater than second string\n";
    cout << "\nStrings before copy:\n" << strOne << endl << strTwo << endl;
    strcpy(strOne,strTwo);
    cout << "\nStrings after copy:\n" << strOne << endl << strTwo << endl;
}
```

Sample run:

```
Enter a string: Joseph
Enter a string: Abraham
First string is greater than second string

Strings before copy:
Jose
Abraham

Strings after copy:
Abraham
Abraham
```

Observations?

It is important to be familiar with C-strings as you will no doubt encounter some C code written using them. The string class however is much easier to use and most prevalent in C++ programs. The string class

- allows strings of unbounded length (recall, the size of a C-string is set at compile time).
- allows aggregate assignment.

- allows aggregate comparison.
- allows concatenation using the + operator as well as other features.

```
void SwapStrings(string& str1, string& str2);
void main(void)
{
     string strOne;
     string strTwo;
     string newString;
     cout << "Enter a string: ";
     getline(cin,strOne);
     cout << "Enter a string: ";
     getline(cin,strTwo);
     cout << "\nFirst String: " << strOne << "\nSecond String: " << strTwo;
     SwapStrings(strOne,strTwo);
     cout << "\n\nAfter swap:" << "\nFirst String: " << strOne << "\nSecond String: "
               << strTwo;
     newString = strOne + " " + strTwo;
     cout << "\n\nThe new string is: " << newString;
}

void SwapStrings(string& str1, string& str2)
{
     string temp;
     temp = str1;
     str1 = str2;
     str2 = temp;
}
```

```
Enter a string: Help me!
Enter a string: Ok, I will help you.

First String: Help me!
Second String: Ok, I will help you.

After swap:
First String: Ok, I will help you.
Second String: Help me!

The new string is: Ok, I will help you. Help me!
```

Observations?

Two-Dimensional Arrays

A one-dimensional array was used to represent a list of items all of the same data type. A two-dimensional array is used to represent a table (rows & columns) of items all of the same type. To access an element in the array you must specify two indexes, one for the row and one for the column.

Declaring a two-dimensional array

GENERAL FORM:

dataType nameOfArray [intExpression] [intExpression];

Note that the intExpression must be a constant.

Examples:

```
int  intArray [5] [5];


const int ROWS = 4;
const int COLS = 5;
float theArray [ROWS] [COLS];
```

Accessing an array element

```
theArray [1] [3] = 12.5;

cout << intArray [2] [3];
```

To load a two-dimensional array you need to use nested loops. Given the following:

```
const int ROWS = 3;
const int COLS = 4;

int myArray[ROWS][COLS];
int r;
int c;

for(r = 0; r < ROWS; r++)
{
    for(c = 0; c < COLS; c++)
     {
          cout << "\nEnter value for row " << r+1 << ",  column " << c+1 << ": ";
          cin >> myArray[r][c];
     }
}
```

Passing Two-Dimensional Arrays as Parameters

When we passed a one-dimensional array to a function it was not necessary to pass the size of the array. Because of the way that two-dimensional arrays are stored in memory, it is ESSENTIAL that the formal parameter list explicitly state the number of columns in the array. The number of rows may be omitted but the number of columns is required. STUDY THE SAMPLE PROGRAM ON THE NEXT PAGE. We will explore another method for declaring arrays when we look at the Typedef statement.

```
// Program to demonstrate the loading of a two-dimensional array
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>

// Wow, a global declaration!
const int NUMOFROWS = 4;
const int NUMOFCOLS = 5;

void LoadSalesData(float sales[][NUMOFCOLS], int rows);
void PrintSalesData(float sales[][NUMOFCOLS], int rows);
void CalcWeeklySales(float sales[][NUMOFCOLS]);

void main(void)
{
    // declaration of a two-dimensional array
    float salesArray[NUMOFROWS][NUMOFCOLS];

    // call function to load the arrays with the month's sales
    LoadSalesData(salesArray,NUMOFROWS);

    // call function to output monthly sales table
    PrintSalesData(salesArray, NUMOFROWS);

    // call function to sum and output weekly sales total
    CalcWeeklySales(salesArray);
}
```

```
void LoadSalesData(float sales[][NUMOFCOLS], int rows)
{
    ifstream inFile;

    // open the file
    inFile.open("sales.dat");

    // load the array
    for(int i = 0; i < rows; i++)
    {
        for(int j = 0; j < NUMOFCOLS; j++)
        {
            inFile >> sales[i][j];
        }
    }
    inFile.close;
}



void PrintSalesData(float sales[][NUMOFCOLS], int rows)
{
    // output headings
    cout << setw(45) << "Monthly Sales\n\n";
    cout << "\t\t\tMonday    Tuesday    Wednesday  Thursday    Friday\n\n";

    // print the array
    for(int i = 0; i < rows; i++)
    {
        cout << "Week " << i+1;
        for(int j = 0; j < NUMOFCOLS; j++)
        {
            cout << setw(12) << sales[i][j];
        }
        cout << endl;
    }
}


void CalcWeeklySales(float sales[][NUMOFCOLS])
{
    for(int i = 0; i < NUMOFROWS; i++)
    {
        float sum;
        sum = 0.0;
        for(int j = 0; j < NUMOFCOLS; j++)
```

```
        {
            sum = sum + sales[i][j];
        }
        cout << "\n\nSales for week " << i+1 << " = " << sum;
    }
}
```

Monthly Sales

	Monday	Tuesday	Wednesday	Thursday	Friday
Week 1	125.50	187.90	110.35	215.75	210.23
Week 2	201.21	215.65	195.65	187.78	275.03
Week 3	95.75	135.55	212.32	127.67	168.92
Week 4	235.82	315.65	486.79	398.45	401.21

Sales for week 1 = 849.73

Sales for week 2 = 1075.32

Sales for week 3 = 740.21

Sales for week 4 = 1837.92