LESSON SET

9 Pointers

PURPOSE

1. To introduce pointer variables and their relationship with arrays
2. To introduce the dereferencing operator
3. To introduce the concept of dynamic memory allocation

PROCEDURE

1. Students should read the Pre-lab Reading Assignment before coming to lab.
2. Students should complete the Pre-lab Writing Assignment before coming to lab.
3. In the lab, students should complete labs assigned to them by the instructor.

Contents	Pre-requisites	Approximate completion time	Page number	Check when done
Pre-lab Reading Assignment		20 min.	158	
Pre-lab Writing Assignment	Pre-lab reading	10 min.	167	
LESSON 9A				
Lab 9.1 Introduction to Pointer Variables	Basic understanding of pointer variables	15 min.	167	
Lab 9.2 Dynamic Memory	Basic understanding of dynamic memory, `new` and `delete` operators	35 min.	168	
LESSON 9B				
Lab 9.3 Dynamic Arrays	Basic understanding of the relationship of pointer variables and arrays	25 min.	170	
Lab 9.4 Student Generated Code Assignments	Basic understanding of pointers, the (*) and (&) symbols, sort and search routines	30 min.	171	

PRE-LAB READING ASSIGNMENT

Pointer Variables

A distinction must always be made between a memory location's address and the data stored at that location. A street address like 119 Main St. is a location that is different than a description of what is at that location: the little red house of the Smith family. So far we have been concerned only with the data stored in a variable, rather than with its address (where in main memory the variable is located). In this lesson we will look at addresses of variables and at special variables, called **pointers**, which hold these addresses. The address of a variable is given by preceding the variable name with the C++ address operator (`&`):

```
cout << &sum; // This outputs the address of the variable sum
```

The & operator in front of the variable `sum` indicates that the address itself, and not the data stored in that location, is the value used. On most systems the above address will print as a hexadecimal value representing the physical location of the variable. Before this lesson where have you used the address operator in C++ programming? You may recall that it was used in the prototype and the function heading of a function for parameters being passed by reference. This connection will be explored in the next section.

To define a variable to be a pointer, we precede it with an asterisk (*) and initialize it with the special value `nullptr`:

```
int *ptr = nullptr;
```

The asterisk in front of the variable indicates that `ptr` holds the address of a memory location. Assigning `nullptr` to a pointer variable makes the variable point to the address 0. When a pointer is set to the address 0, it is referred to as a null pointer because it points to "nothing." The `int` indicates that the memory location that `ptr` points to holds integer values. `ptr` is NOT an integer data type, but rather a pointer that holds the address of a location where an integer value is stored. This distinction is most important!

The following example illustrates this difference.

```
int  sum;                  // sum holds an integer value.
int *sumPtr = nullptr;     // sumPtr holds an address where an
                           // integer can be found.
```

By now there may be confusion between the symbols `*` and `&`, so we next discuss their use.

Using the & Symbol

The `&` symbol is basically used on two occasions.

1. The most frequent use we have seen is between the data type and the variable name of a pass by reference parameter in a function heading/prototype. This is called a **reference variable**. The memory address of the parameter is sent to the function instead of the value at that address. When the parameter is used in the function, the compiler automatically **dereferences** the variable. Dereference means that the location of that reference variable (parameter in this case) is accessed to retrieve or store a value.

We have looked at the swap function on several occasions. We revisit this routine to show that the & symbol is used in the parameters that need to be swapped. The reason is that these values need to be changed by the function and, thus, we give the address (location in memory) of those values so that the function can write their new values into them as they are swapped.

Example:

```
void swap(int &first, int &second)
{                   // The & indicates that the parameters
                    // first and second are being passed by
                    // reference.

int temp;

temp = first;     // Since first is a reference variable,
                  // the compiler retrieves the value
                  // stored there and places it in temp.

first = second    // New values are written directly into
second = temp;    // the memory locations of first and second.
}
```

2. The & symbol is also used whenever we are interested in the *address* of a variable rather than its *contents*.

Example:

```
cout  << sum;     // This outputs the value stored in the
                  // variable sum.
cout  << &sum;    // This outputs the address where
                  // sum is stored in memory.
```

Using the & symbol to get the address of a variable comes in handy when we are assigning values to pointer variables.

Using the * Symbol

The * symbol is also basically used on two occasions.

1. It is used to define pointer variables:

```
int *ptr = nullptr;
```

2. It is also used whenever we are interested in the contents of the *memory location* pointed to by a *pointer variable*, rather than the address itself. When used this way * is called the **indirection operator**, or **dereferencing operator**.

Example:

```
cout << *ptr; // Since ptr is a pointer variable, *
              // dereferences ptr. The value stored at the
              // location ptr points to will be printed.
```

Using * and & Together

In many ways * and & are the opposites of each other. The * symbol is used just before a pointer variable so that we may obtain the actual data rather than the address of the variable. The & symbol is used on a variable so that the variable's address, rather than the data stored in it, will be used. The following program demonstrates the use of pointers.

Sample Program 9.1:

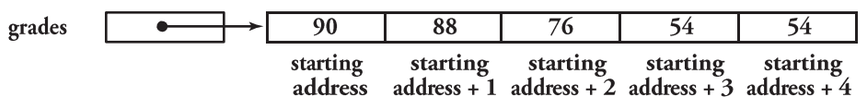
```
#include <iostream>
using namespace std;

int main()
{
   int one = 10;
   int *ptr1 = nullptr;  //  ptr1 is a pointer variable that points to an int

   ptr1 = &one;    //  &one indicates that the address, not the
                   //  contents, of one is being assigned to ptr1.
                   //  Remember that ptr1 can only hold an address.
                   //  Since ptr1 holds the address where the variable
                   //  one is stored, we say that ptr1 "points to" one.

   cout << "The value of one is   " << one   << endl << endl;
   cout << "The value of &one is  " << &one  << endl << endl;
   cout << "The value of ptr1 is  " << ptr1  << endl << endl;
   cout << "The value of *ptr1 is " << *ptr1 << endl << endl;

   return 0;
}
```

What do you expect will be printed if the address of variable one is the hexadecimal value 006AFOF4? The following will be printed by the program.

Output	Comments
The value of one is 10	one is an integer variable, holding a 10.
The value of &one is 006AF0F4	&one is the "address of" variable one.
The value of ptr1 is 006AF0F4	ptr1 is assigned one's address
The value of *ptr1 is 10	* is the dereferencing operator which means *ptr1 gives us the value of the variable ptr1 is pointing at.

Arrays and Pointers

When arrays are passed to functions they are passed by pointer. An array name is a pointer to the beginning of the array. Variables can hold just one value and so we can reference that value by just naming the variable. Arrays, however, hold many values. All of these values cannot be referenced just by naming the array. This is where pointers enter the picture. Pointers allow us to access all the array elements. Recall that the array name is really a pointer that holds the address of the first element in the array. By using an array index, we dereference the pointer which gives us the contents of that array location. If `grades` is an array of 5 integers, as shown below, `grades` is actually a pointer to the first location in the array, and `grades[0]` allows us to access the contents of that first location.



From the last section we know it is also possible to dereference the pointer by using the * operator. What is the output of the following two statements?

```
cout << grades[0];   //  Output the value stored in the 1st array element
cout << *grades;     //  Output the value found at the address stored
                     //  in grades (i.e., at the address of the 1st array
                     //  element).
```

Both statements are actually equivalent. They both print out the contents of the first `grades` array location, a 90.

Access of an individual element of an array through an index is done by **pointer arithmetic**. We can access the second array location with `grades[1]`, the third location with `grades[2]`, and so on, because the indices allow us to move through memory to other addresses relative to the beginning address of the array. The phrase "address + 1" in the previous diagram means to move one array element forward from the staring address of the array. The third element is accessed by moving 2 elements forward and so forth. The amount of movement in bytes depends on how much memory is allocated for each element, and that depends on how the array is defined. Since `grades` is defined as an array of integers, if an integer is allocated 4 bytes, then +1 means to move forward 4 bytes from the starting address of the array, +2 means to move forward 8 bytes, etc. The compiler keeps track of how far forward to move to find a desired element based on the array index. Thus the following two statements are equivalent.

```
cout << grades[2];
cout << *(grades + 2);
```

Both statements refer to the value located two elements forward from the starting address of the array. Although the first may be the easiest, computer scientists need to understand how to access memory through pointers. The following program illustrates how to use pointer arithmetic rather than indexing to access the elements of an array.

Sample Program 9.2:

```
// This program illustrates how to use pointer arithmetic to
// access elements of an array.

#include <iostream>
using namespace std;

int main()
{
    int grades[] = {90, 88, 76, 54, 34};
        // This defines and initializes an int array.
        // Since grades is an array name, it is really a pointer
        // that holds the starting address of the array.

    cout << "The first grade is "          // The * before grades
         << *grades << endl;               // dereferences it so that the
                                           // contents of array location 0
                                           // is printed instead of its
                                           // address.

    cout << "The second grade is "         // The same is done for the other
         << *(grades + 1) << endl;         // elements of the array. In
    cout << "The third grade is "          // each case, pointer arithmetic
         << *(grades + 2) << endl;         // gives us the address of the
    cout << "The fourth grade is "         // next array element.  Then the
         << *(grades + 3) << endl;         // indirection operator * gives
    cout << "The fifth grade is "          // us the value of what is stored
         << *(grades + 4) << endl;         // at that address.

    return 0;
}
```

What is printed by the program?

```
The first grade is 90
The second grade is 88
The third grade is 76
The fourth grade is 54
The fifth grade is 34
```

Dynamic Variables

In Lesson Set 7 on arrays, we saw how the size of an array is given at the time of its definition. The programmer must estimate the maximum number of elements that will be used by the array and this size is static, i.e., it cannot change during the execution of the program. Consequently, if the array is defined to be larger than is needed, memory is wasted. If it is defined to be smaller than is needed, there is not enough memory to hold all of the elements. The use of pointers (and the `new` and `delete` operators described below) allows us to dynamically allocate enough memory for an array so that memory is not wasted.

This leads us to **dynamic variables**. Pointers allow us to use dynamic variables, which can be created and destroyed as needed within a program. We have studied scope rules, which define where a variable is active. Related to this is the concept of **lifetime**, the time during which a variable exists. The lifetime of dynamic variables is controlled by the program through explicit commands to allocate (i.e., create) and deallocate (i.e., destroy) them. The operator **`new`** is used to allocate and the operator **`delete`** is used to deallocate dynamic variables. The compiler keeps track of where in memory non-dynamic variables (variables discussed thus far in this book) are located. Their contents can be accessed by just naming them. However, the compiler does not keep track of the address of a dynamic variable. When the `new` command is used to allocate memory for a dynamic variable, the system returns its address and the programmer stores it in a pointer variable. Through the pointer variable we can access the memory location.

Example:

```
int *one = nullptr;    // one and two are defined to be pointer
int *two = nullptr;    // variables that point to ints

int result;       // defines an int variable that will hold
                  // the sum of two values.

one = new int;    // These statements each dynamically
two = new int;    // allocate enough memory to hold an int
                  // and assign their addresses to pointer
                  // variables one and two, respectively.

*one = 10;        // These statements assign the value 10
*two = 20;        // to the memory location pointed to by one
                  // and 20 to the memory location pointed to
                  // by two.

result = *one + *two;
                  // This adds the contents of the memory
                  // locations pointed to by one and two.
cout << "result = " << result << endl;

delete one;       // These statements deallocate the dynamic
delete two;       // variables.  Their memory is freed and
                  // they cease to exist.
```

Now let us use dynamic variables to allocate an appropriate amount of memory to hold an array. By using the `new` operator to create the array, we can wait until we know how big the array needs to be before creating it. The following program demonstrates this idea. First the user is asked to input the number of grades to be processed. Then that number is used to allocate exactly enough memory to hold an array with the required number of elements for the grades.

Sample Program 9.3:

```
// This program finds the average of a set of grades.
// It dynamically allocates space for the array holding the grades.

#include <iostream>
#include <iomanip>
using namespace std;

// function prototypes
void sortIt (float* grades, int numOfGrades);
void displayGrades(float* grades, int numOfGrades);

int main()
{
    float *grades = nullptr;  // a pointer that will be used to point
                              // to the beginning of a float array
    float total = 0;          // total of all grades
    float average;            // average of all grades
    int numOfGrades;          // the number of grades to be processed
    int count;                // loop counter

    cout << fixed << showpoint << setprecision(2);

    cout << "How many grades will be processed " << endl;
    cin >> numOfGrades;

    while (numOfGrades <= 0)  // checks for a legal value
    {
       cout << "There must be at least one grade. Please reenter.\n";
       cout << "How many grades will be processed " << endl;
       cin >> numOfGrades;
    }

    grades = new float(numOfGrades);
                           // allocation memory for an array
                           // new is the operator that is allocating
                           // an array of floats with the number of
                           // elements specified by the user. grades
                           // is the pointer holding the starting
                           // address of the array.

    if (grades == nullptr)  // nullptr is a special identifier
    {                       // to equal 0.  It indicates a non-valid
                            // address.  If grades is 0 it means the
                            // the operating system was unable to
                            // allocate enough memory for the array.

    cout << "Error allocating memory!\n";
                          // The program should output an appropriate
    return -1;            // error message and return with a value
    }                     // other than 0 to signal a problem.
    cout << "Enter the grades below\n";
```

```
    for (count = 0; count < numOfGrades; count++)
    {
        cout << "Grade " << (count + 1) << ": " << endl;
        cin >> grades[count];
        total = total + grades[count];
    }

    average = total / numOfGrades;
    cout << "Average Grade is " << average << "%" << endl;

    sortIt(grades, numOfGrades);
    displayGrades(grades, numOfGrades);
    delete [] grades;           // deallocates all the array memory

    return 0;
}

//*************************************************
//                sortIt
//
// task:          to sort numbers in an array
// data in:       an array of floats and
//                the number of elements in the array
// data out:      sorted array
//
//*************************************************

void sortIt(float* grades, int numOfGrades)
{
  // Sort routine placed here
}

//*************************************************
//                displayGrades
//
// task:          to display numbers in an array
// data in:       an array of floats and
//                the number of elements in the array
// data out:      none
//
//*************************************************
void displayGrades(float* grades, int numOfGrades)
{
       // Code to display grades of the array
}
```

Notice how the dynamic array is passed as a parameter to the `sortIt` and `displayGrades` functions. In each case, the call to the function simply passes the name of the array, along with its size as an argument. The name of the array holds the array's starting address.

```
sortIt(grades, numOfGrades);
```

In the function header, the formal parameter that receives the array is defined to be a pointer data type.

```
void sortIt(float* grades, int numOfGrades)
```

Since the compiler treats an array name as a pointer, we could also have written the following function header.

```
void sortIt(float grades[], int numOfGrades)
```

In this program, dynamic allocation of memory was used to save memory. This is a minor consideration for the type of programs done in this course, but a major concern in professional programming environments where large fluctuating amounts of data are used.

Review of * and &

The * symbol is used to define pointer variables. In this case it appears in the variable definition statement between the data type and the pointer variable name. It indicates that the variable holds an address, rather than the data stored at that address.

Example 1: `int *ptr1;`

* is also used as a dereferencing operator. When placed in front of an already defined pointer variable, the data stored at the location the pointer points to will be used and not the address.

Example 2: `cout << *ptr1;`

Since `ptr1` is defined as a pointer variable in Example 1, if we assume `ptr1` has now been assigned an address, the output of Example 2 will be the data stored at that address. `*` in this case dereferences the variable `ptr1`.

The `&` symbol is used in a procedure or function heading to indicate that a parameter is being passed by reference. It is placed between the data type and the parameter name of each parameter that is passed by reference.

The `&` symbol is also used before a variable to indicate that the address, not the contents, of the variable is to be used.

Example 3:

```
int *ptr1 = nullptr;
int one = 10;

ptr1 = &one;            // This assigns the address of variable
                        // one to ptr1

cout << "The value of &one is "
     << &one << endl;  // This prints an address

cout << "The value of *ptr1 is "
     << *ptr1 << endl;  // This prints 10, because ptr1 points to
                        // one and * is the dereferencing operator.
```

PRE-LAB WRITING ASSIGNMENT

Fill-in-the-Blank Questions

1. The _____symbol is the dereferencing operator.
2. The _____symbol means "address of."
3. The name of an array, without any brackets, acts as a(n) _____ to the starting address of the array.
4. An operator that allocates a dynamic variable is _____.
5. An operator that deallocates a dynamic variable is _____.
6. Parameters that are passed by _____are similar to a pointer variable in that they can contain the address of another variable. They are used as parameters of a procedure (void function) whenever we want a procedure to change the value of the argument.

Given the following information, fill the blanks with either "an address" or "3.75".

```
float * pointer;
float pay = 3.75;
pointer = &pay;
```

7. `cout << pointer;` will print _____.
8. `cout << *pointer;` will print _____.
9. `cout << &pay;` will print _____.
10. `cout << pay;` will print _____.

LESSON 9A

LAB 9.1 Introduction to Pointer Variables

Retrieve program `pointers.cpp` from the Lab 9 folder.
The code is as follows:

```
// This program demonstrates the use of pointer variables
// It finds the area of a rectangle given length and width
// It prints the length and width in ascending order

// PLACE YOUR NAME HERE

#include <iostream>
using namespace std;

int main()
{

   int length;                // holds length
   int width;                 // holds width
   int area;                  // holds area

   int *lengthPtr = nullptr;  // int pointer which will be set to point to length
   int *widthPtr = nullptr;   // int pointer which will be set to point to width
```

continues

```
   cout << "Please input the length of the rectangle" << endl;
   cin >> length;
   cout << "Please input the width of the rectangle" << endl;
   cin >> width;

   // Fill in code to make lengthPtr point to length (hold its address)
   // Fill in code to make widthPtr point to width (hold its address)


   area =  // Fill in code to find the area by using only the pointer variables
   cout << "The area is " << area << endl;


   if (// Fill in the condition length > width by using only the pointer variables)
      cout << "The length is greater than the width" << endl;

   else if (// Fill in the condition of width > length by using only the pointer
            // variables)
      cout << "The width is greater than the length" << endl;

   else
      cout << "The width and length are the same" << endl;


   return 0;
}

}
```

Exercise 1: Complete this program by filling in the code (places in bold). Note: use only pointer variables when instructed to by the comments in bold. This program is to test your knowledge of pointer variables and the `&` and `*` symbols.

Exercise 2: Run the program with the following data: 10 15. Record the output here _____.

LAB 9.2 Dynamic Memory

Retrieve program `dynamic.cpp` from the Lab 9 folder. The code is as follows:

```
// This program demonstrates the use of dynamic variables
// PLACE YOUR NAME HERE

#include <iostream>
using namespace std;

const int MAXNAME = 10;
```

```
int main()
{
   int pos;
   char *name = nullptr;
   int *one = nullptr;
   int *two = nullptr;
   int *three = nullptr;
   int result;

   //  Fill in code to allocate the integer variable one here
   //  Fill in code to allocate the integer variable two here
   //  Fill in code to allocate the integer variable three here
   //  Fill in code to allocate the character array pointed to by name

   cout << "Enter your last name with exactly 10 characters." << endl;
   cout << "If your name has < 10 characters, repeat last letter. " << endl
        << "Blanks at the end do not count." << endl;

   for (pos = 0;  pos < MAXNAME;   pos++)

      cin >>   // Fill in code to read a character into the name array
               // WITHOUT USING a bracketed subscript

   cout << "Hi ";
   for (pos = 0;  pos < MAXNAME;  pos++)

      cout << // Fill in code to a print a character from the name array
              // WITHOUT USING a bracketed subscript

   cout << endl << "Enter three integer numbers separated by blanks" << endl;

   // Fill in code to input three numbers and store them in the
   // dynamic variables pointed to by pointers one, two, and three.
   // You are working only with pointer variables

   //echo print
   cout << "The three numbers are " << endl;

   // Fill in code to output those numbers

   result = // Fill in code to calculate the sum of the three numbers
   cout << "The sum of the three values is " << result << endl;

   // Fill in code to deallocate one, two, three and name

   return 0;
}
```

Exercise 1: Complete the program by filling in the code. (Areas in bold) This problem requires that you study very carefully the code already written to prepare you to complete the program.

Sample Run:

```
Enter your last name with exactly 10 characters.
If your name < 10 characters, repeat last letter. Blanks do not count.
DeFinooooo
Hi DeFinooooo
Enter three integer numbers separated by blanks
5 6 7
The three numbers are 5 6 7
The sum of the three values is 18
```

Exercise 2: In inputting and outputting the name, you were asked NOT to use a bracketed subscript. Why is a bracketed subscript unnecessary?

Would using `name[pos]` work for inputting the name? Why or why not? Would using `name[pos]` work for outputting the name? Why or why not?

Try them both and see.

LESSON 9B

LAB 9.3 Dynamic Arrays

Retrieve program `darray.cpp` from the Lab 9 folder. The code is as follows:

```
// This program demonstrates the use of dynamic arrays

// PLACE YOUR NAME HERE

#include <iostream>
#include <iomanip>
using namespace std;


int main()
{
   float *monthSales = nullptr;   // a pointer used to point to an array
                                  // holding monthly sales
   float  total = 0;              // total of all sales
   float  average;                // average of monthly sales
   int    numOfSales;             // number of sales to be processed
   int    count;                  // loop counter

   cout << fixed << showpoint << setprecision(2);

   cout << "How many monthly sales will be processed? ";
   cin >> numOfSales;

   // Fill in the code to allocate memory for the array pointed to by
   // monthSales.
```

```
   if ( // Fill in the condition to determine if memory has been
        // allocated (or eliminate this if construct if your instructor
        // tells you it is not needed for your compiler)
      )

   {
         cout << "Error allocating memory!\n";
         return 1;
   }

   cout << "Enter the sales below\n";

   for (count = 0; count < numOfSales;  count++)
   {
       cout << "Sales for Month number  "
            << // Fill in code to show the number of the month
            << ":";

       // Fill in code to bring sales into an element of the array
   }

   for (count = 0; count < numOfSales; count++)
   {
       total = total + monthSales[count];
   }

   average = // Fill in code to find the average


   cout << "Average Monthly sale is $" << average << endl;
   // Fill in the code to deallocate memory assigned to the array.

   return 0;
}
```

Exercise 1: Fill in the code as indicated by the comments in bold.

Sample Run:

```
How many monthly sales will be processed 3
Enter the sales below
Sales for Month number 1: 401.25
Sales for Month number 2: 352.89
Sales for Month number 3: 375.05
Average Monthly sale is $376.40
```

LAB 9.4 Student Generated Code Assignments

In these assignments you are asked to develop functions that have dynamic arrays as parameters. Remember that dynamic arrays are accessed by a pointer variable and thus the parameters that serve as dynamic arrays are, in fact, pointer variables.

Example:

```
void sort(float* score, int num_scores); // a prototype whose function has a
                                         // dynamic array as its first
                                         // parameter. It is a pointer variable
.
.
int main()
{
   float *score = nullptr;               // a pointer variable
     .
     .
   score  =  new float(num_scores);      // allocation of the array

   sort(score,scoreSize);                // call to the function
```

Option 1: Write a program that will read scores into an array. The size of the array should be input by the user (dynamic array). The program will find and print out the average of the scores. It will also call a function that will sort (using a bubble sort) the scores in ascending order. The values are then printed in this sorted order.

Sample Run:

```
Please input the number of scores
5
Please enter a score
100
Please enter a score
90
Please enter a score
95
Please enter a score
100
Please enter a score
90
The average of the scores is 95

Here are the scores in ascending order
90
90
95
100
100
```

Option 2: This program will read in id numbers and place them in an array. The array is dynamically allocated large enough to hold the number of id numbers given by the user. The program will then input an id and call a function to search for that id in the array. It will print whether the id is in the array or not.

Sample Run:

```
Please input the number of id numbers to be read
4
Please enter an id number
96
Please enter an id number
97
Please enter an id number
98
Please enter an id number
99

Please input an id number to be searched
67
67 is not in the array
```

Option 3: Write a program that will read monthly sales into a dynamically allocated array. The program will input the size of the array from the user. It will call a function that will find the yearly sum (the sum of all the sales). It will also call another function that will find the average.

Sample Run:

```
Please input the number of monthly sales to be input
4
Please input the sales for month 1
1290.89
Please input the sales for month 2
905.95
Please input the sales for month 3
1567.98
Please input the sales for month 4
994.83
The total sales for the year is $4759.65
The average monthly sale is $1189.91
```

LESSON SET

10 Characters and Strings

PURPOSE

1. To demonstrate the unique characteristics of character data
2. To view strings as an array of characters
3. To show how to input and output strings
4. To work with string functions

PROCEDURE

1. Students should read the Pre-lab Reading Assignment before coming to the lab.
2. Students should complete the Pre-lab Writing Assignment before coming to lab.
3. In the lab, students should complete labs assigned to them by the instructor.

Contents	Pre-requisites	Approximate completion time	Page number	Check when done
Pre-lab Reading Assignment		20 min.	176	
Pre-lab Writing Assignment	Pre-lab reading	10 min.	186	
LESSON 10A				
Lab 10.1 Character Testing and String Validation	Pre-lab reading	15 min.	187	
Lab 10.2 Case Conversion	Basic fundamental instructions	5 min.	190	
Lab 10.3 Using `getline() & get()`	Basic knowledge of character arrays	30 min.	192	
LESSON 10B				
Lab 10.4 String Functions—`strcat`	Basic knowledge of character arrays	15 min.	193	
Lab 10.5 Student Generated Code Assignments	Basic knowledge of character arrays	35 min.	193	

PRE-LAB READING ASSIGNMENT

Character Functions

C++ provides numerous *functions* for character testing. These functions will test a single character and return either a non-zero value (true) or zero (false). For example, `isdigit` tests a character to see if it is one of the digits between 0 and 9. So `isdigit(7)` returns a non-zero value whereas `isdigit(y)` and `isdigit($)` both return 0. We will not list all the character functions here. A complete list may be found in the text. The following program demonstrates some of the others. Note that the `cctype` header file must be included to use the character functions.

Sample Program 10.1:

```
// This program utilizes several functions for character testing

#include <iostream>
#include <cctype>
using namespace std;

int main()
{
    char input;

    cout << "Please Enter Any Character:" << endl;
    cin >> input;
    cout << "The character entered is " << input << endl << endl;
    cout << "The ASCII code for " << input << " is " << int(input)
         << endl;

    if (isalpha(input))     // tests to see if character is a letter
    {
      cout << "The character is a letter" << endl;

      if (islower(input))   // tests to see if letter is lower case
             cout << "The letter is lower case" << endl;

      if (isupper(input))   // tests to see if letter is upper case
             cout << "The letter is upper case" << endl;
    }

    else if (isdigit(input)) // tests to see if character is a digit
             cout << "The character you entered is a digit" << endl;

    else
             cout << "The character entered is not a letter nor a digit"
                  << endl;

    return 0;
}
```

In Lab 10.1 you will see a more practical application of character testing functions.

Character Case Conversion

The C++ library provides the `toupper` and `tolower` functions for converting the case of a character. `toupper` returns the uppercase equivalent for a letter and `tolower` returns the lower case equivalent. For example, `cout << tolower('F');` causes an `f` to be displayed on the screen. If the letter is already lowercase, then `tolower` will return the value unchanged. Likewise, any non-letter argument is returned unchanged by `tolower`. It should be clear to you now what `toupper` does to a given character.

While the `toupper` and `tolower` functions are conceptually quite simple, they may not appear to be very useful. However, the following program shows that they do have beneficial applications.

Sample Program 10.2:

```
// This program shows how the toupper and tolower functions can be
// applied in a C++ program

#include <iostream>
#include <cctype>
#include <iomanip>
using namespace std;

int main()
{
   int week, total, dollars;
   float average;
   char choice;

   cout << showpoint << fixed << setprecision(2);

   do
   {
      total = 0;
      for(week = 1; week <= 4; week++)
      {
             cout << "How much (to the nearest dollar) did you"
                  << " spend on food during week " << week
                   << " ?:" << endl;

             cin >> dollars;

             total = total + dollars;
      }
      average = total / 4.0;

      cout << "Your weekly food bill over the chosen month is $"
           << average << endl << endl;
      do
      {
             cout << "Would you like to find the average for "
                  << "another month?";
```

continues

```
            cout << endl << "Enter Y or N" << endl;
            cin >> choice;
        } while(toupper(choice) != 'Y' && toupper(choice) != 'N');

    } while (toupper(choice) == 'Y');

    return 0;
}
```

This program prompts the user to input weekly food costs, to the nearest dollar (an integer) for a four-week period. The average weekly total for that month is output. Then the user is asked whether they want to repeat the calculation for a different month. The flow of this program is controlled by a `do-while` loop. The condition `toupper(choice) == 'Y'` allows the user to enter `'Y'` or `'y'` for yes. This makes the program more user friendly than if we just allowed `'Y'`. Note the second `do-while` loop near the end of the program. This loop also utilizes `toupper`. Can you determine the purpose of this second loop? How would the execution of the program be affected if we removed this loop (but left in the lines between the curly brackets)?

String Constants

We have already talked about the character data type which includes letters, digits, and other special symbols such as $ and @. Often we need to put characters together to form strings. For example, the price "$1.99" and the phrase "one for the road!" are both strings of characters. The phrase contains blank space characters in addition to letters and an exclamation mark. In C++ a string is treated as a sequence of characters stored in consecutive memory locations. The end of the string in memory is marked by the null character `\0`. Do not confuse the null character with a sequence of two characters (i.e., \ and 0). The null character is actually an escape sequence. Its ASCII code is 0. For example, the phrase above is stored in computer memory as

o	n	e		f	o	r		t	h	e		r	o	a	d	!	\0

A **string constant** is a string enclosed in double quotation marks. For example,

"Learn C++"
"What time is it?"
"Code Word 7dF#c&Q"

are all string constants. When they are stored in the computer's memory, the null character is automatically appended. The string "Please enter a digit" is stored as

P	l	e	a	s	e		e	n	t	e	r		a		d	i	g	i	t	\0

When a string constant is used in C++, it is the memory address that is actually accessed. In the statement

```
cout << "Please enter a digit";
```

the memory address is passed to the `cout` object. `cout` then displays the consecutive characters until the null character is reached.

Storing Strings in Arrays

Often we need to access parts of a string rather than the whole string. For instance, we may want to alter characters in a string or even compare two strings. If this is the case, then a string constant is not what we need. Rather, a character array is the appropriate choice. When using character arrays, enough space to hold the null character must be allocated. For example:

```
char last[10];
```

This code defines a 10-element character array called `last`. However, this array can hold no more than 9 non-null characters since a space is reserved for the null character. Consider the following:

```
char last[10];
cout << "Please enter your last name using no more than 9 letters";
cin >> last;
```

If the user enters `Symon`, then the following will be the contents of the `last` array:

S	y	m	o	n	\0

Recall that the computer actually sees `last` as the beginning address of the array. There is a problem that can arise when using the `cin` object on a character array. `cin` does not "know" that `last` has only 10 elements. If the user enters `Newmanouskous` after the prompt, then `cin` will write past the end of the array. We can get around this problem by using the `getline` function. If we use

```
cin.getline(last,10)
```

then the computer knows that the maximum length of the string, including the null character, is 10. Consequently, `cin` will read until the user hits `ENTER` or until 9 characters have been read, whichever occurs first. Once the string is in the array, it can be processed character by character. In this next section we will see a program that uses `cin.getline()`.

Library Functions for Strings

The C++ library provides many functions for testing and manipulating strings. For example, to determine the length of a given string one can use the `strlen` function. The syntax is shown in the following code:

```
char line[40] = "A New Day";
int length;
length = strlen(line);
```

Here `strlen(line)` returns the length of the string including white spaces but not the null character at the end. So the value of `length` is 9. Note this is smaller than the size of the actual array holding the string.

To see why we even need a function such as `strlen`, consider the problem of reading in a string and then writing it backwards. If we only allowed strings of a fixed size, say length 29 for example, then the task would be easy. We simply read the string into an array of size 30 or more. Then write the 28^{th} entry followed by the 27^{th} entry and so on, until we reach the 0^{th} entry. However, what if we wish to allow the user to input strings of different lengths? Now it is unclear where the end of the string is. Of course, we could search the array until we find

the null character and then figure out what position it is in. But this is precisely what the `strlen` function does for us. Sample Program 10.3 is a complete program that performs the desired task.

Sample Program 10.3:

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
       char line[50];
       int length,count = 0;

       cout << "Enter a sentence of no more than 49 characters:\n";
       cin.getline(line,50);

       length = strlen(line);    // strlen returns the length of the
                                 // string currently stored in line

       cout << "The sentence entered read backwards is:\n";

       for(count = length-1; count >= 0; count--)
       {
             cout << line[count];

       }

    cout << endl;
    return 0;
}
```

Sample Run 1:

```
Enter a sentence of no more than 49 characters:
luaP deiruB I
The sentence you entered printed backwards is:
I Buried Paul
```

Sample Run 2:

```
Enter a sentence of no more than 49 characters:
This sentence is too long to hold a mere 49 characters!
The sentence you entered printed backwards is:
arahc 94 erem a dloh ot gnol oot si ecnetnes sihT
```

Another useful function for strings is `strcat`, which concatenates two strings. `strcat(string1,string2)` attaches the contents of `string2` to the end of `string1`. The programmer must make sure that the array containing `string1` is large enough to hold the concatenation of the two strings plus the null character.

Consider the following code:

```
char string1[25] = "Total Eclipse ";     // note the space after the second
                                         // word - strcat does not insert a
                                         // space. The programmer must do this.
char string2[11] = "of the Sun";

cout << string1 << endl;
cout << string2 << endl;

strcat(string1,string2);

cout << string1 << endl;
```

These statements produce the following output:

```
Total Eclipse
of the Sun
Total Eclipse of the Sun
```

What would have happened if we had defined `string1` to be a character array of size 20?

There are several other string functions such as `strcpy` (copies the second string to the first string), `strcmp` (compares two strings to see if they are the same or, if not, which string is alphabetically greater than the other), and `strstr` (looks for the occurrence of a string inside of another string). Note that C-string functions require the `cstring` header file. For more details on these string functions and the others, see the text.

The `get` and `ignore` functions

There are several ways of inputting strings. We could use the standard >> extraction operator for a character array or string class object. However, we know that using `cin >>` skips any leading whitespace (blanks, newlines). It will also stop at the first trailing whitespace character. So, for example, the name "John Wayne" cannot be read as a single string using `cin >>` because of a blank space between the first and last names. We have already seen the `getline` function which does allow blank spaces to be read and stored. In this section we will introduce the `get` and `ignore` functions, which are also useful for string processing.

The `get` function reads in the next character in the input stream, including whitespace. The syntax is

```
cin.get(ch);
```

Once this function call is made, the next character in the input stream is stored in the variable `ch`. So if we want to input

```
$ X
```

we can use the following:

```
cin.get(firstChar);
cin.get(ch);
cin.get(secondChar);
```

where `firstChar`, `ch`, and `secondChar` are all character variables. Note that after the second call to the `get` function, the blank character is stored in the variable `ch`.

The `get` function, like the `getline` function, can also be used to read strings. In this case we need two parameters:

```
cin.get(strName, numChar+1);
```

Here `strName` is a string variable and the integer expression `numChar+1` gives the number of characters that may be read into `strName`.

Both the `getline` and the `get` functions do not skip leading whitespace characters. The `get` statement above brings in the next input characters until it either has read `numChar+1` characters or it reaches the newline character `\n`. However, the newline character is not stored in `strName`. The null character is then appended to the end of the string. Since the newline character is not **`consumed`** (not read by the `get` function), it remains part of the input characters yet to be read.

Example:

```
char strName[21];
cin.get(strName,21);
```

Now suppose we input

John Wayne

Then "John Wayne" is stored in `strName`. Next input

My favorite westerns star John Wayne

In this case the string "My favorite westerns" is stored in `strName`.

We often work with records from a file that contain character data followed by numeric data. Look at the following data which has a name, hours worked, and pay rate for each record stored on a separate line.

Pay Roll Data

```
John Brown          7   12.50
Mary Lou Smith     12   15.70
Dominic DeFino      8   15.50
```

Since names often have imbedded blank spaces, we can use the `get` function to read them. We then use an integer variable to store the number of hours and a floating point variable to store the pay rate. At the end of each line is the '`\n`' character. Note that the end of line character is not consumed by reading the pay rate and, in fact, is the next character to be read when reading the second name from the file. This creates problems. Whenever we need to read through characters in the input stream without storing them, we can use the `ignore` function. This function has two arguments, the first is an integer expression and the second is a character expression. For example, the call

```
cin.ignore(80,'\n');
```

says to skip over the next 80 input characters but stop if a newline character is read. The newline character is consumed by the `ignore` function. This use of `ignore` is often employed to find the end of the current input line.

The following program will read the sample pay roll data from a file called payRoll.dat and show the result to the screen. Note that the input file must have names that are no longer than 15 characters and the first 15 positions of each line are reserved for the name. The numeric data must be after the 15th position in each line.

Sample Program 10.4:

```
#include <fstream>
#include <iostream>
using namespace std;

const int MAXNAME = 15;

int main()
{
      ifstream inData;

      inData.open("payRoll.dat");
      char name[MAXNAME+1];
      int hoursWorked;
      float payRate;

      inData.get(name,MAXNAME+1);  // prime the read
      while (inData)
      {
        inData >> hoursWorked;
        inData >> payRate;

        cout << name << endl;
        cout << "Hours Worked " << hoursWorked << endl;
        cout << "Pay Rate " << payRate << " per hour"
             << endl << endl;

        inData.ignore(80,'\n');
         // This will ignore up to 80 characters but will
         // stop (ignoring) when it reads the \n which is
         // consumed.

        inData.get(name,MAXNAME+1);

      }

      return 0;
}
```

Summary of types of input for strings:

```
cin >> strName;         // skips leading whitespace. Stops at the first
                        // trailing whitespace (which is not consumed)

cin.get(strName, 21);   // does not skip leading whitespace
                        // stops when either 20 characters are read or
                        // '\n' is encountered (which is not consumed)

cin.ignore(200,'\n');   // ignores at most 200 characters but stops if
                        // newline (which is consumed) is encountered
```

Pointers and Strings

Pointers can be very useful for writing string processing functions. If one needs to process a certain string, the beginning address can be passed with a pointer variable. The length of the string does not even need to be known since the computer will start processing using the address and continue through the string until the null character is encountered.

Sample Program 10.5 below reads in a string of no more than 50 characters and then counts the number of letters, digits, and whitespace characters in the string. Notice the use of the pointer `strPtr`, which points to the string being processed. The three functions `countLetters`, `countDigits`, and `countWhiteSpace` all perform basically the same task—the `while` loop is executed until `strPtr` points to the null character marking the end of the string. In the `countLetters` function, characters are tested to see if they are letters. The `if(isalpha(*strPtr))` statement determines if the character pointed at by `strPtr` is a letter. If so, then the counter `occurs` is incremented by one. After the character has been tested, `strPtr` is incremented by one to test the next character. The other two functions are analogous.

Sample Program 10.5:

```
#include <iostream>
#include <cctype>

using namespace std;

//function prototypes
int countLetters(char*);
int countDigits(char*);
int countWhiteSpace(char*);

int main()
{
     int numLetters, numDigits, numWhiteSpace;
     char inputString[51];


     cout <<"Enter a string of no more than 50 characters: "
          << endl << endl;
```

```
    cin.getline(inputString,51);

    numLetters = countLetters(inputString);
    numDigits = countDigits(inputString);
    numWhiteSpace = countWhiteSpace(inputString);

    cout << "The number of letters in the entered string is "
         << numLetters << endl;
    cout << "The number of digits in the entered string is "
         << numDigits << endl;
    cout << "The number of white spaces in the entered string is "
         << numWhiteSpace << endl;

    return 0;
}
//*****************************************************************
//                   countLetters
//
// task:              This function counts the number of letters
//                    (both capital and lower case) in the string
// data in:           pointer that points to an array of characters
// data returned:     number of letters in the array of characters
//
//*****************************************************************

int countLetters(char *strPtr)
{
    int occurs = 0;
    while(*strPtr != '\0')       // loop is executed as long as
                                 // the pointer strPtr does not point
                                 // to the null character which
                                 // marks the end of the string
    {
       if (isalpha(*strPtr))     // isalpha determines if
                                 // the character is a letter
             occurs++;
       strPtr++;
    }
    return occurs;
}
//*****************************************************************
//                    countDigits
//
// task:              This function counts the number of digits
//                    in the string
// data in:           pointer that points to an array of characters
// data returned:     number of digits in the array of characters
//
//*****************************************************************
```

continues

```
int countDigits(char *strPtr)
{
    int occurs = 0;
    while(*strPtr != '\0')
    {
       if (isdigit(*strPtr))    // isdigit determines if
                                // the character is a digit
          occurs++;
       strPtr++;
    }
    return occurs;
}

//*****************************************************************
//                    countWhiteSpace
//
// task:              This function counts the number of whitespace
//                    characters in the string
// data in:           pointer that points to an array of characters
// data returned:     number of whitespaces in the array of
//                    characters
//
//*****************************************************************

int countWhiteSpace(char *strPtr)   // this function counts the
                                    // number of whitespace characters.
                                    // These include, space, newline,
                                    // vertical tab, and tab
{
    int occurs = 0;
    while(*strPtr != '\0')
    {
       if (isspace(*strPtr))        // isspace determines if
                                    // the character is a
                                    // whitespace character
             occurs++;
       strPtr++;
    }
    return occurs;
}
```

PRE-LAB WRITING ASSIGNMENT

Fill-in-the-Blank Questions

1. The code `cout << toupper('b');` causes a _____ to be displayed on the screen.
2. The data type returned by `isalpha('g')` is _____.

3. After the assignment statement `result = isdigit('$')`, `result` has the value _____.
4. The code `cout << tolower('#');` causes a _____to be displayed on the screen.
5. The end of a string is marked in computer memory by the _____.
6. In `cin.getline(name,25)`, the 25 indicates that the user can input at most _____characters into `name`.
7. Consider the following:

```
char message[35] = "Like tears in the rain";
int length;
length = strlen(message);
```

   Then the value of `length` is _____.

8. Consider the code

```
char string1[30] = "In the Garden";
char string2[15] = "of Eden";
strcat(string1,string2);
cout << string1;
```

   The output for this is _____.

9. The _____header file must be included to access the `islower` and `isspace` character functions.
10. In C++, a string constant must be enclosed in _____whereas a character constant must be enclosed in _____.

LESSON 10

LAB 10.1 Character Testing and String Validation

The American Equities investment company offers a wide range of investment opportunities ranging from mutual funds to bonds. Investors can check the value of their portfolio from the American Equities' web page. Information about personal portfolios is protected via encryption and can only be accessed using a password. The American Equities company requires that a password consist of 8 characters, 5 of which must be letters and the other 3 digits. The letters and digits can be arranged in any order. For example,

rt56AA7q
123actyN
1Lo0Dwa9
myNUM741

are all valid passwords. However, the following are all invalid:

the476NEw	// It contains more than 8 characters (also more than 5 // letters)
be68moon	// It contains less than 3 digits.
$retrn99	// It contains only 2 digits and has an invalid character ('$')

American Equities needs a program for their web page that determines whether or not an entered password is valid. The program `american_equities.cpp` from the Lab 10 folder performs this task. The code is the following:

```
// This program tests a password for the American Equities
// web page to see if the format is correct

// Place Your Name Here

#include <iostream>
#include <cctype>
#include <cstring>

using namespace std;

//function prototypes

bool testPassWord(char[]);
int countLetters(char*);
int countDigits(char*);

int main()
{
      char passWord[20];

      cout << "Enter a password consisting of exactly 5 "
           << "letters and 3 digits:" << endl;
      cin.getline(passWord,20);

      if (testPassWord(passWord))
           cout << "Please wait - your password is being verified" << endl;
      else
      {
           cout << "Invalid password. Please enter a password "
                << "with exactly 5 letters and 3 digits" << endl;
           cout << "For example, my37RuN9 is valid" << endl;
      }

       // Fill in the code that will call countLetters and
       // countDigits and will print to the screen both the number of
       // letters and digits contained in the password.

       return 0;
}
```

```
//*************************************************************
//                      testPassWord
//
// task:             determines if the word in the
//                   character array passed to it, contains
//                   exactly 5 letters and 3 digits.
// data in:          a word contained in a character array
// data returned:    true if the word contains 5 letters & 3
//                   digits, false otherwise
//
//*************************************************************
bool testPassWord(char custPass[])
{
   int numLetters, numDigits, length;

   length = strlen(custPass);
   numLetters = countLetters(custPass);
   numDigits = countDigits(custPass);
   if (numLetters == 5 && numDigits == 3 && length == 8 )
      return true;
   else
      return false;
}

// the next 2 functions are from Sample Program 10.5
//*************************************************************
//                        countLetters
//
// task:             counts the number of letters (both
//                   capital and lower case)in the string
// data in:          a string
// data returned:    the number of letters in the string
//
//*************************************************************
int countLetters(char *strPtr)
{
   int occurs = 0;

   while(*strPtr != '\0')

   {
      if (isalpha(*strPtr))
         occurs++;
      strPtr++;
   }

   return occurs;
}
```

continues

```
//*************************************************************
//                       countDigits
//
// task:               counts the number of digits in the string
// data in:            a string
// data returned:      the number of digits in the string
//
//*************************************************************
int countDigits(char *strPtr)
{
    int occurs = 0;

    while(*strPtr != '\0')
    {
        if (isdigit(*strPtr))    // isdigit determines if
                                 // the character is a digit
            occurs++;
        strPtr++;
    }

    return occurs;
}
```

Exercise 1: Fill in the code in bold and then run the program several times with both valid and invalid passwords. Read through the program and make sure you understand the logic of the code.

Exercise 2: Alter the program so that a valid password consists of 10 characters, 6 of which must be digits and the other 4 letters.

Exercise 3: Adjust your program from Exercise 2 so that only lower case letters are allowed for valid passwords.

LAB 10.2 Case Conversion

Bring in `case_convert.cpp` from the Lab 10 folder. Note that this is Sample Program 10.2. The code is the following:

```
// This program shows how the toupper and tolower functions can be
// applied in a C++ program

// PLACE YOUR NAME HERE

#include <iostream>
#include <cctype>
#include <iomanip>
using namespace std;

int main()
{
    int week, total, dollars;
    float average;
    char choice;
```

```
    cout << showpoint << fixed << setprecision(2);

    do
    {
        total = 0;
        for(week = 1; week <= 4; week++)
        {
            cout << "How much (to the nearest dollar) did you"
                 << " spend on food during week " << week
                 << " ?:" << endl;
            cin  >> dollars;

            total = total + dollars;
        }
        average = total / 4.0;

        cout << "Your weekly food bill over the chosen month is $"
             << average << endl << endl;
        do
        {
           cout << "Would you like to find the average for "
                << "another month?";
           cout << endl << "Enter Y or N" << endl;
           cin >> choice;
        } while(toupper(choice) != 'Y' && toupper(choice) != 'N');

    } while (toupper(choice) == 'Y');


    return 0;
}
```

Exercise 1: Run the program several times with various inputs.

Exercise 2: Notice the following `do-while` loop which appears near the end of the program:

```
do
{
   cout << "Would you like to find the average for another month?";
   cout << endl << "Enter Y or N" << endl;
   cin >> choice;
} while(toupper(choice) != 'Y' && toupper(choice) != 'N');
```

How would the execution of the program be different if we removed this loop? Try removing the loop but leave the following lines in the program:

```
cout << "Would you like to find the average for another month?";
cout << endl << "Enter Y or N" << endl;
cin >> choice;
```

Record what happens when you run the new version of the program.

Exercise 3: Alter program `case_convert.cpp` so that it performs the same task but uses `tolower` rather than `toupper`.

LAB 10.3 Using `getline() & get()`

Exercise 1: Write a short program called `readata.cpp` that defines a character array `last` which contains 10 characters. Prompt the user to enter their last name using no more than 9 characters. The program should then read the name into `last` and then output the name back to the screen with an appropriate message. Do not use the `getline()` or `get` functions!

Exercise 2: Once the program in Exercise 1 is complete, run the program and enter the name **Newmanouskous** at the prompt. What, if anything, happens? (Note that the results could vary depending on your system).

Exercise 3: Re-write the program above using the `getline()` function (and only allowing 9 characters to be input). As before, use the character array `last` consisting of 10 elements. Run your new program and enter **Newmanouskous** at the prompt. What is the output?

Exercise 4: Bring in program `grades.cpp` and `grades.txt` from the Lab 10 folder. Fill in the code in bold so that the data is properly read from `grades.txt`. and the desired output to the screen is as follows:

OUTPUT TO SCREEN			DATA FILE	
Adara Starr	**has a(n)**	**94 average**	**Adara Starr**	**94**
David Starr	**has a(n)**	**91 average**	**David Starr**	**91**
Sophia Starr	**has a(n)**	**94 average**	**Sophia Starr**	**94**
Maria Starr	**has a(n)**	**91 average**	**Maria Starr**	**91**
Danielle DeFino	**has a(n)**	**94 average**	**Danielle DeFino**	**94**
Dominic DeFino	**has a(n)**	**98 average**	**Dominic DeFino**	**98**
McKenna DeFino	**has a(n)**	**92 average**	**McKenna DeFino**	**92**
Taylor McIntire	**has a(n)**	**99 average**	**Taylor McIntire**	**99**
Torrie McIntire	**has a(n)**	**91 average**	**Torrie McIntire**	**91**
Emily Garrett	**has a(n)**	**97 average**	**Emily Garrett**	**97**
Lauren Garrett	**has a(n)**	**92 average**	**Lauren Garrett**	**92**
Marlene Starr	**has a(n)**	**83 average**	**Marlene Starr**	**83**
Donald DeFino	**has a(n)**	**73 average**	**Donald DeFino**	**73**

The code of `grades.cpp` is as follows:

```
#include <fstream>
#include <iostream>
using namespace std;

// PLACE YOUR NAME HERE

const int MAXNAME = 20;

int main()
{
    ifstream inData;
    inData.open("grades.txt");

    char name[MAXNAME + 1];  // holds student name
    float average;           // holds student average
```

```
    inData.get(name,MAXNAME+1);
    while (inData)
    {
        inData >> average;
        // Fill in the code to print out name and
        // student average

        // Fill in the code to complete the while
        // loop so that the rest of the student
        // names and average are read in properly
    }

    return 0;
}
```

LAB 10.4 String Functions—`strcat`

Consider the following code:

```
char string1[25] ="Total Eclipse ";
char string2[11] = "of the Sun";
cout << string1 << endl;
cout << string2 << endl;
strcat(string1,string2);
cout << string1 << endl;
```

Exercise 1: Write a complete program including the above code that outputs the concatenation of `string1` and `string2`. Run the program and record the result.

Exercise 2: Alter the program in Exercise 1 so that `string1` contains 20 characters rather than 25. Run the program. What happens?

LAB 10.5 Student Generated Code Assignments

Exercise 1: A **palindrome** is a string of characters that reads the same forwards as backwards. For example, the following are both palindromes:

1457887541 madam

Write a program that prompts the user to input a string of a size 50 characters or less. Your program should then determine whether or not the entered string is a palindrome. A message should be displayed to the user informing them whether or not their string is a palindrome.

Exercise 2: The `strcmp(string1,string2)` function compares `string1` to `string2`. It is a value returning function that returns a negative integer if `string1 < string2`, 0 if `string1 == string2`, and a positive integer if `string1 > string2`. Write a program that reads two names (last name first followed by a comma followed by the first name) and then prints them in alphabetical order. The two names should be stored in separate character arrays holding a maximum of 25 characters each. Use the `strcmp()` function to make the comparison of the two names. Remember that `'a' < 'b'`, `'b' < 'c'`, etc. Be sure to include the proper header file to use `strcmp()`.

Sample Run 1:

Please input the first name
Brown, George

Please input the second name
Adams, Sally

The names are as follows:
Adams, Sally
Brown, George

Sample Run 2:

Please input the first name
Brown, George

Please input the second name
Brown, George

The names are as follows:
Brown, George
Brown, George
The names are the same

Exercise 3: (Optional) Write a program that determines how many consonants are in an entered string of 50 characters or less. Output the entered string and the number of consonants in the string.