Lecture 8, 9

Lecture 8

Objects and Classes

8.1 Const Member Functions

A const member function guarantees that it will never modify any of its class's member data. The following example shows how this works.

```
class aClass
{
      private:
            int alpha;
      public:
            void nonFunc()              //non-const member function
            { alpha = 99; }             //OK
            void conFunc() const        //const member function
            { alpha = 99; }             //ERROR: can't modify a member
};
```

The non-const function *nonFunc()* can modify member data alpha, but the constant function *conFunc()* can't. If it tries to, a compiler error results. A function is made into a constant function by placing the keyword *const* after the declarator but before the function body. If there is a separate function declaration, const must be used in both declaration and definition. Member functions that do nothing but acquire data from an object are obvious candidates for being made const, because they don't need to modify any data.

8.2 Exercises

1. In a class definition, data or functions designated private are accessible
   a. to any function in the program.
   b. only if you know the password.
   c. to member functions of that class.
   d. only to public members of the class.

2. True or false: Data items in a class must be private.

3. The dot operator (or class member access operator) connects the following two entities (reading from left to right):

   a. A class member and a class object

   b. A class object and a class

   c. A class and a member of that class

   d. A class object and a member of that class

4. Member functions defined inside a class definition are ________ by default.

5. A constructor is executed automatically when an object is ________.

6. A constructor's name is the same as _________.

7. True or false: In a class you can have more than one constructor with the same name.

8. A member function can always access the data

   a. in the object of which it is a member.

   b. in the class of which it is a member.

   c. in any object of the class of which it is a member.

   d. in the public part of its class.

9. The only technical difference between structures and classes in C++ is that ________.

10. For the object for which it was called, a const member function

   a. can modify both const and non-const member data.

   b. can modify only const member data.

   c. can modify only non-const member data.

   d. can modify neither const nor non-const member data.

11. Create a class called time that has separate int member data for hours, minutes, and seconds. One constructor should initialize this data to 0, and another should initialize it to fixed values. Another member function should display it, in 11:59:59 format. The final member function should add two objects of type time passed as arguments. A main() program should create two initialized time objects (should they be const?) and one that isn't initialized. Then it should add the two initialized values together, leaving the result in the third time variable. Finally it should display the value of this third variable. Make appropriate member functions const.

arrayNamelen

strlen(str1)

Lecture 9

Arrays

9.1 Single-Dimensional Arrays

9.1.1 Declaring Single-Dimensional Arrays

C++ requires that you declare an array before you use it. The general syntax for declaring an array is:

type arrayName[*numberOfElements*];

This syntax shows the following aspects of the declaration:

1. The declaration starts by stating the basic type associated with the array elements. You can use predefined or previously defined data types.
2. The name of the array is followed by the number of elements. This number is enclosed in square brackets. The number of array elements must be a constant (literal or symbolic) or an expression that uses constants.

All arrays in C++ have indices that start at 0. Thus, the number of array elements is one larger than the index of the last array element. Here are examples of declaring arrays:

```
// example 1
  int nIntArr[10];
// example 2
  const int MAX = 30;
  char cName[MAX];
// example 3
  const int MAX_CHARS = 40;
  char cString[MAX_CHARS+1];
```

The first example declares the **int**-type array **nIntArr** with 10 elements. The declaration uses the literal constant 10. Thus the indices for the first and last array elements are 0 and 9, respectively. The second example declares the constant **MAX** and uses that constant to specify the number of elements of the **char**-type array **cName**. The third example declares the **char**-type array **cString**. The constant expression **MAX_CHARS + 1** defines the number of elements in the array **cString**.

9.1.2 Accessing Single-Dimensional Arrays

Once you declare an array, you can access its elements using the index operator **[]**. The general syntax for accessing an element in an array is:

arrayName[*anIndex*]

The index should be in the valid range of indices–between 0 and the number of array elements minus one. Here are some examples of accessing array elements:

```
const int MAX = 10;
double fVector[MAX];
for (int i = 0; i < MAX; i++)
  fVector[i] = double(i) * i;
for (i = MAX - 1; i >= 0; i--)
  cout << fVector[i] << "\n";
```

This code snippet declares the constant **MAX** and uses that constant in declaring the **double**-type array **fVector**. Thus the array has elements with indices in the range of 0 to **MAX — 1**. The code uses the first **for** loop to assign values to the elements of array **fVector**. The loop statement accesses the elements of array **fVector** using the loop control variable **i**. The expression **fVector[i]** accesses element number **i** in array **fVector**. The code uses the second loop to display, in a descending order, the elements of array **fVector**. Again, the loop statement accesses the elements of array **fVector** using the loop control variable **i**.

9.1.3 Initializing Single-Dimensional Arrays

C++ allows you to initialize some or all of the elements of an array. The general syntax for initializing an array is:

type arrayName[*numOfElems*] = { *value0* ,..., *valueN* };

You need to observe the following rules when you initialize an array:

1. The list of initial values appears in a pair of open and close braces and is comma-delimited. The list ends with a semicolon.
2. The list may contain a number of initial values that is equal to or less than the number of elements in the initialized array. Otherwise, the compiler generates a compile-time error.
3. The compiler assigns the first initializing value to the element at index 0, the second initializing value to the element at index 1, and so on.
4. If the list contains fewer values than the number of elements in the array, the compiler assigns zeros to the elements that do not receive initial values from the list.
5. If you omit the number of array elements, the compiler uses the number of initializing values in the list as the number of array elements.

Here are examples of initializing arrays:

```
// example 1
      double fArr[5] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
// example 2
```

```
        int nArr[10] = { 1, 2, 3, 4, 5 };
    // example 3
        char cVowels[] = { 'A' , 'a', 'E', 'e', 'I', 'i', 'O', 'o', 'U', 'u' };
```

The code in example 1 declares the **double**-type array **fArr** to have 5 elements. The declaration also initializes all 5 elements with the successive values 1.1, 2.2, 3.3, 4.4, and 5.5. The second example declares the **int**-type array **nArr** to have 10 elements. The declaration also initializes only the first 5 elements with the values 1, 2, 3, 4, and 5. Therefore, the compiler assigns zeros to the elements at indices 5 through 9 of array **nArr**. The last example declares the **char**-type array **cVowels** and initializes the array elements with the lowercase and uppercase vowels. Since the declaration of array **cVowels** does not specify the number of elements, the compiler uses the number of initializing values, 10, as the number of elements in array **cVowels**.

9.1.4 Declaring Arrays as Function Parameters

C++ allows you to declare arrays as parameters to functions. The syntax for declaring such parameters is:

```
// form 1: fixed parameter
  type arrayParam[numberOfElements]
// form 2: open parameter
  type openArrayParam[]
```

The first form specifies the number of elements. Use this form when you want an array of the specified size to be the argument for the parameter **arrayParam**. This kind of parameter is called a *fixed array* parameter. The second form does not specify the size of the array parameter, allowing the parameter to take arguments that are arrays of different sizes. This kind of parameter is called an *open array* parameter. In this case, you typically have an additional parameter that specifies the number of elements in the parameter **openArrayParam**. Here are examples of declaring array parameters:

```
double mySum(double fMyArray[MAX_ELEMS]);
double theirSum(double fTheirArray[MAX_ELEMS],
            int nNumElems = MAX_ELEMS);
double yourSum(double fYourArray[], int nNumElems);
```

The function **mySum** declares the array parameter **fMyArray** and specifies the size of **MAX_ELEMS**. This kind of function expects the caller to pass a **double**-type array that has **MAX_ELEMS** number of elements. Since the function's parameter list does not have a parameter that passes the number of elements to process, it is safe to assume that function **mySum** processes all of the elements in parameter **fMyArray**.

The function **theirSum** has two parameters. The first is the array parameter **fMyArray** and specifies the size of **MAX_ELEMS**. The second is **nNumElems**. The parameters of the function **theirSum** suggest that the arguments for parameter **fTheirArray** must be **double**-type arrays with **MAX_ELEMS** elements. However, the presence of parameter **nNumElems** may suggest that the function can process a portion of the array **fTheirArray**.

The function **yourSum** declares the parameter **fYourArray**, which is an open **double**-type array. This function can take arguments that are **double**-type arrays of different sizes. The parameter **nNumElems** specifies the number of elements in the argument for parameter **fYourArray**. The argument for parameter **nNumElems** should be equal to or less than the number of elements in the argument for **fYourArray**. If not, the function risks accessing data that lie beyond the space occupied by the array argument.

Here are examples of calling the functions just described:

```
const int MAX_ELEMS = 10;
const int MAX_ARRAY = 20;

double fArray1[MAX_ELEMS];
double fArray2[MAX_ARRAY];
double fSum;
...
fSum = mySum(fArray1);
...
fSum = theirSum(fArray1);
...
fSum = theirSum(fArray1, MAX_ELEMS / 2);
...
fSum = yourSum(fArray1, MAX_ELEMS);
...
fSum = yourSum(fArray2, MAX_ARRAY);
```

This code snippet declares the constants **MAX_ELEMS** and **MAX_ARRAY**, which define the number of elements in the **double**-type arrays **fArray1** and **fArray2**, respectively. The code defines the **double**-type variable **fSum**. Then the code snippet makes the following calls to the tested functions:

1. Call function **mySum** with the argument **fArray1**

2. Call function **theirSum** with the argument **fArray1**. This function call uses the default argument of **MAX_ELEMS** for parameter **nNumElems**.
3. Call function **theirSum** with the arguments **fArray1** and **MAX_ELEMS / 2**
4. Call function **yourSum** with the arguments **fArray1** and **MAX_ELEMS**
5. Call function **yourSum** with the arguments **fArray2** and **MAX_ARRAY**

The code snippet shows that function **yourSum** is very flexible, since it handles arrays of different sizes.

9.2 Multidimensional Arrays

9.2.1 Declaring Multidimensional Arrays

C++ requires that you declare a multidimensional array before you use it. The general syntax for declaring a multidimensional array is:

type arrayName[*numberOfElement1*][*numberOfElement2*];

The above syntax shows the following aspects:

1. The declaration starts by stating the basic type associated with the array elements. You can use predefined or previously defined data types.
2. The name of the array is followed by a sequence of the number of elements for the various dimensions. These numbers appear in square brackets. Each number of elements must be a constant (literal or symbolic) or an expression that uses constants.

All multidimensional arrays in C++ have indices that start at 0. Thus, the number of array elements in each dimension is one value higher than the index of the last element in that dimension.

Here are examples of declaring multidimensional arrays:

```
// example 1
        int nIntCube[20][10][5];
// example 2
        const int MAX_ROWS = 50;
        const int MAX_COLS = 20;
        double fMatrix[MAX_ROWS][MAX_COLS];
// example 3
        const int MAX_ROWS = 30;
        const int MAX_COLS = 10;
        char cNameArray[MAX_ROWS+1][MAX_COLS];
```

The first example declares the **int**-type three-dimensional array **nIntCube** with 20 by 10 by 5 elements. The declaration uses the literal constants 20, 10, and 5. Thus, the indices for the first dimension are in the range of 0 to 19, the indices for the second dimension are in the range of 0 to 9, and the indices for the third dimension are in the range of 0 to 4.

The second example declares the constants **MAX_ROWS** and **MAX_COLS** and uses these constants to specify the number of rows and columns of the **double**-type matrix **fMatrix.**

The third example declares the **char**-type matrix **cNameArray**. The constant expression **MAX_ROWS + 1** defines the number of rows in the array **cNameArray**. The constant **MAX_COLS** defines the number of columns in the array **cNameArray**.

9.2.2 Accessing Multidimensional Arrays

Once you declare a multidimensional array, you can access it elements using the index operator **[]**. The general syntax for accessing an element in a multidimensional array is:

arrayName[IndexOfDimension1][IndexOfDimension2]...

The indices for the various dimensions should be in the valid ranges—between 0 and the number of array elements for the dimension minus one. Here is an example of accessing multidimensional array elements:

```
const int MAX_ROWS = 10;
const int MAX_COLS = 20;
double fMatrix[MAX_ROWS][MAX_COLS];
for (int i = 0; i < MAX_ROWS; i++)
  for (int j = 0; j < MAX_COLS; j++)
    fMatrix[i][j] = double(2 + i+2 * j)
```

This code snippet declares the constants **MAX_ROWS** and **MAX_COLS** and uses these constants in declaring the **double**-type two-dimensional array **fMatrix**. Thus the array has rows with indices in the range of 0 to **MAX_ROWS — 1** and columns with indices in the range of 0 to **MAX_COLS — 1**. The code snippet uses nested **for** loops to initialize the elements of the array **fMatrix**. Notice that the last assignment statement uses the syntax **fMatrix[i][j]** and not **fMatrix[i, j]** or **fMatrix(i, j)** as is the case in Pascal and BASIC, respectively.

9.2.3 Initializing Multidimensional Arrays

C++ allows you to initialize some or all of the elements of a multidimensional array. The general syntax for initializing a multidimensional array is:

type arrayName[numberOfElement1][numberOfElement2] = { value0 ,..., valueN };

You need to observe the following rules when you initialize a multidimensional array:

1. The list of initial values appears in a pair of open and close braces and is comma-delimited.
2. The list may contain a number of initial values that is equal to or less than the total number of elements in the initialized array. Otherwise, the compiler generates a compile-time error.
3. The compiler assigns the initializing values in the sequence discussed in the sidebar "Initializing Multidimensional Arrays."
4. If the list contains fewer values than the number of elements in the array, the compiler assigns zeros to the elements that do not receive initial values from the list.

Here are examples of initializing multidimensional arrays:

```
// example 1
  double fMat[3][2] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
// example 2
  int nMat[5][2] = { 1, 2, 3, 4, 5 };
// example 3
  double fMat[3][2] = {{ 1.1, 2.2}, {3.3, 4.4}, {5.5, 6.6} };
```

The code in example 1 declares the **double**-type two-dimensional array **fMat** to have three rows and two columns. The declaration also initializes all six elements with the values 1.1, 2.2, 3.3, 4.4, 5.5, and 6.6. The compiler stores these values sequentially in elements **fMat[0][0]**, **fMat[0][1]**, and so on. The second example declares the **int**-type matrix **nMat** to have five rows and two columns. The declaration also initializes only the first five elements (**nMat[0][0]**, **nMat[0][1]**, **nMat[1][0]**, **nMat[1][1]**, and **nMat[2][0]**) with the values 1, 2, 3, 4, and 5. Therefore, the compiler assigns zeros to the remaining matrix elements. The third example is similar to the second example but it initializes the multi-dimensional array as an array of arrays. The initializing values for each subarray are enclosed in braces and separated by commas

9.2.4 Declaring Multidimensional Arrays as Function Parameters

C++ allows you to declare multidimensional arrays as parameters in functions. The programming language supports two syntaxes, one for fixed arrays and the other one for open arrays:

```
// fixed array parameter
  type parameterName[numberOfElement1][numberOfElement2]
// open-array parameter
  type parameterName[][numberOfElement2]
```

The fixed-array parameter states the number of elements for each dimension. By contrast, the open array parameter lists the number of elements for the second dimension and up. In other words, the open parameter leaves the number of elements for the first dimension unspecified.

Here are examples of declaring multidimensional array parameters:

```
double mySum(double fMyMatrix[MAX_ROWS1][MAX_COLS]);
double theirSum(double fTheirMatrix[MAX_ROWS2][MAX_COLS],
            int nNumRows = MAX_ROWS1,
            int nNumCols = MAX_COLS);
double yourSum(double fYourMatrix[][MAX_COLS],
           int nNumRows, int nNumCols);
```

The function **mySum** declares the matrix parameter **fMyMatrix** and specifies that the parameter has **MAX_ROWS1** rows and **MAX_COLS** columns. This kind of function expects the caller to pass a

double-type matrix that also has **MAX_ROWS1** rows and **MAX_COLS** columns. Since the function's parameter list does not have parameters that pass the number of rows and columns to process, it is safe to assume that function **mySum** processes all of the elements in parameter **fMyMatrix**.

The function **theirSum** has three parameters. The first one is the matrix parameter **fMyMatrix** and specifies **MAX_ROWS2** rows and **MAX_COLS** columns. The second and third parameters, **nNumRows** and **nNumCols**, specify the number of rows and columns, respectively, to process. Thus, the first parameter suggests that the arguments for parameter **fTheirMatrix** must be **double**-type matrices with **MAX_ROWS2** rows and **MAX_COLS** columns. However, the presence of parameters **nNumRows** and **nNumCols** might suggest that the function can process a portion of the matrix **fTheirMatrix**.

The function **yourSum** declares the parameter **fYourMatrix**, which is an open **double**-type matrix. This function can take arguments that are **double**-type matrices of different numbers of rows. However, the arguments for the matrix parameter must have **MAX_COLS** columns. The parameters **nNumRows** and **nNumCols** specify the number of matrix rows and columns, respectively, to process. The argument for parameter **nNumRows** should be equal to or less than the number of rows in the argument for **fYourMatrix**. If not, the function risks accessing data that lie beyond the space occupied by the matrix argument.

Here are examples of calling these functions:

```
        const int MAX_ROWS1 = 10;
```

```
const int MAX_ROWS2 = 10;
const int MAX_COLS = 20;

double fMatrix1[MAX_ROWS1][MAX_COLS];
double fMatrix2[MAX_ROWS2][MAX_COLS];
double fSum;
...
fSum = mySum(fMatrix1);
...
fSum = theirSum(fMatrix2);
...
fSum = theirSum(fMatrix2, MAX_ROWS2 / 2, MAX_COLS / 2);
...
fSum = yourSum(fMatrix1, MAX_ROWS1, MAX_COLS);
...
fSum = yourSum(fMatrix2, MAX_ROWS2, MAX_COLS);
```

This code snippet declares the constants **MAX_ROWS1**, **MAX_ROWS2**, and **MAX_COLS**, which define the number of rows and columns in the **double**-type matrices **fMatrix1** and **fMatrix2**, respectively. The code defines the **double**-type variable **fSum**. It then makes the following calls to the tested functions:

1. Call function **mySum** with the argument **fMatrix1**
2. Call function **theirSum** with the argument **fMatrix2**. This function call uses the default arguments of **MAX_ROWS2** and **MAX_COLS** for parameters **nNumRows** and **nNumCols**, respectively.
3. Call function **theirSum** with the arguments **fMatrix2**, **MAX_ROWS2 / 2, MAX_COLS / 2**
4. Call function **yourSum** with the arguments **fMatrix1**, **MAX_ROWS1**, and **MAX_COLS**

5. Calls function **yourSum** with the arguments **fMatrix2, MAX_ROWS2,** and **MAX_COLS**

The code snippet shows that function **yourSum** is very flexible, since it handles matrices of different row sizes.

9.3 Exercises

1. Answer each of the following:

a) Lists and tables of values are stored in __________.

b) The elements of an array are related by the fact that they have the same __________and __________.

c) The number used to refer to a particular element of an array is called its __________.

d) A __________should be used to declare the size of an array, because it makes the program more scalable.

e) An array that uses two subscripts is referred to as a __________array.

2. State whether the following are true or false. If the answer is false, explain why.

a) An array can store many different types of values.

b) An array subscript should normally be of data type float.

c) If there are fewer initializers in an initializer list than the number of elements in the array, the remaining elements are automatically initialized to the last value in the list of initializers.

d) It is an error if an initializer list contains more initializers than there are elements in the array.

e) An individual array element that is passed to a function and modified in that function will contain the modified value when the called function completes execution.

3. Answer the following questions regarding an array called fractions :

a) Define a constant variable arraySize initialized to 10.

b) Declare an array with arraySize elements of type double, and initialize the elements to 0.

c) Name the fourth element from the beginning of the array.

d) Refer to array element 4.

e) Assign the value 1.667 to array element 9.

f) Assign the value 3.333 to the seventh element of the array.

g) Print array elements 6 and 9 with two digits of precision to the right of the decimal point, and show the output that is actually displayed on the screen.

h) Print all the elements of the array using a for repetition structure. Define the integer variable x as a control variable for the loop. Show the output.

4. Answer the following questions regarding an array called table:

a) Declare the array to be an integer array and to have 3 rows and 3 columns. Assume that the constant variable arraySize has been defined to be 3.

b) How many elements does the array contain?

c) Use a for repetition structure to initialize each element of the array to the sum of its subscripts. Assume that the integer variables x and y are declared as control variables.

d) Write a program segment to print the values of each element of an array table in tabular format with 3 rows and 3 columns. Assume that the array was initialized with the declaration and the integer variables x and y are declared as control variables. Show the output.

```
int table[ arraySize ][ arraySize ] =
{{ 1, 8 }, { 2, 4, 6 }, { 5 } };
```

and the integer variables x and y are declared as control variable. Show the output.

5. Find the error in each of the following program segments and correct the error:

a) #include <iostream>;

b) arraySize = 10; //arraySize was declared const

c) Assume that

```
int b[ 10 ] = {0 };
for ( int i = 0; i <= 10; i++ )
  b[i ] = 1;
```

d) Assume that

```
int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
a[ 1, 1 ] = 5;
```

6. Fill in the blanks in each of the following:
   a) C++ stores lists of values in __________.
   b) The elements of an array are related by the fact that they __________.
   c) When referring to an array element, the position number contained within square brackets is called a __________.
   d) The names of the four elements of array p are__________ , __________, __________ and __________.
   e) Naming an array, stating its type and specifying the number of elements in the array is called __________ the array.
   f) The process of placing the elements of an array into either ascending or descending order is called __________.
   g) In a double-subscripted array, the first subscript (by convention) identifies the __________ of an element, and the second subscript (by convention) identifies the __________ of an element.
   h) An m-by-n array contains__________ rows, __________ columns and __________ elements.
   i) The name of the element in row 3 and column 5 of array d is __________.

7. State which of the following are true and which are false; for those that are false, explain why they are false.
   a) To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element.
   b) An array declaration reserves space for the array.
   c) To indicate that 100 locations should be reserved for integer array p, the programmer writes the declaration p[100];

d) A C++ program that initializes the elements of a 15-element array to zero must contain at least one for statement.

e) A C++ program that totals the elements of a double-subscripted array must contain nested for statements.

8. Write C++ statements to accomplish each of the following:

a) Display the value of the seventh element of character array f.

b) Input a value into element 4 of single-subscripted floating-point array b.

c) Initialize each of the 5 elements of single-subscripted integer array g to 8.

d) Total and print the elements of floating-point array c of 100 elements.

e) Copy array a into the first portion of array b. Assume double a[11], b[34];

f) Determine and print the smallest and largest values contained in 99-element floating-point array w.

9. Consider a 2-by-3 integer array t.

a) Write a declaration for t.

b) How many rows does t have?

c) How many columns does t have?

d) How many elements does t have?

e) Write the names of all the elements in the second row of t.

f) Write the names of all the elements in the third column of t.

g) Write a single statement that sets the element of t in row 1 and column 2 to zero.

h) Write a series of statements that initialize each element of t to zero. Do not use a loop.

i) Write a nested for structure that initializes each element of t to zero.

j) Write a statement that inputs the values for the elements of t from the terminal.

k) Write a series of statements that determine and print the smallest value in array t.

l) Write a statement that displays the elements of the first row of t.

m) Write a statement that totals the elements of the fourth column of t.

n) Write a series of statements that prints the array t in neat, tabular format. List the column subscripts as headings across the top and list the row subscripts at the left of each row.

10. Write single statements that perform the following single-subscripted array operations:
   a) Initialize the 10 elements of integer array counts to zero.
   b) Add 1 to each of the 15 elements of integer array bonus.
   c) Read 12 values for double array monthlyTemperatures from the keyboard.
   d) Print the 5 values of integer array bestScores in column format.

11. Find the error(s) in each of the following statements:
   a) Assume that: char str[5];
      cin >> str; // User types hello
   b) Assume that: int a[3];
      cout << a[1] << " " << a[2] << " " << a[3] << endl;
   c) double f[3] = { 1.1, 10.01,100.001, 1000.0001 };
   d) Assume that: double d[2][10];
      d[1, 9] = 2.345;

12. Use a single-subscripted array to solve the following problem. Read in 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, print it only if it is not a duplicate of a number already read.

13. Use a double-subscripted array to solve the following problem. A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each different type of product sold. Each slip contains the following:
   a) The salesperson number
   b) The product number
   c) The total dollar value of that product sold that day

14. (Print an array) Write a recursive function printArray that takes an array and the size of the array as arguments and returns nothing. The function should stop processing and return when it receives an array of size zero.

15. (Print a string backwards) Write a recursive function stringReverse that takes a character array containing a string as an argument, prints the string backwards and returns nothing. The function should stop processing and return when the terminating null character is encountered.

16. (Find the minimum value in an array) Write a recursive function recursiveMinimum that takes an integer array and the array size as arguments and returns the smallest element of the array. The function should stop processing and return when it receives an array of 1 element.

Lecture 10

C-Strings

10.1 C-String Variables

In this lecture. C-strings are decsribed as arrays of type char. The following example example defines a single C-string variable. It asks the user to enter a string, and places this string in the string variable. Then it displays the string.

```
#include <iostream.h>
int main()
{
        const int MAX = 80;          //max characters in string
        char str[MAX];               //string variable str

        cout << "Enter a string: ";
        cin >> str;                  //put string in str
                                     //display string from str
        cout << "You entered: " << str << endl;
        return 0;
}
```

10.2 Avoiding Buffer Overflow

It is possible to tell the >> operator to limit the number of characters it places in an array to avoid inserting array elements outside an array.

```
#include <iostream.h>
#include <iomanip.h>                 //for setw
int main()
{
        const int MAX = 20;          //max characters in string
        char str[MAX];                //string variable str
        cout << "\nEnter a string: ";
        cin >> setw(MAX) >> str;     //put string in str,
                                     // no more than MAX chars
        cout << "You entered: " << str << endl;
        return 0;
}
```

10.3 String Constants

You can initialize a string to a constant value when you define it. Here's an example:

```
#include <iostream.h>
int main()
{
        char str[] = "Farewell! thou art too dear for my possessing.";
        cout << str << endl;
        return 0;
}
```

10.4 Reading Embedded Blanks

To read text containing blanks we use another function, cin.get(). The following example shows how it's used.

```
#include <iostream.h>
int main()
{
        const int MAX = 80;           //max characters in string
        char str[MAX];                //string variable str
        cout << "\nEnter a string: ";
        cin.get(str, MAX);            //put string in str
        cout << "You entered: " << str << endl;
        return 0;
}
```

10.5 Reading Multiple Lines

The cin::get() function can take a third argument. This argument specifies the character that tells the function to stop reading. The default value for this argument is the newline ('\n') character, but if you call the function with some other character for this argument, the default will be overridden by the specified character. In the next example we call the function with a dollar sign ('$') as the third argument:

```
#include <iostream.h>
const int MAX = 2000;                 //max characters in string
char str[MAX];                        //string variable str
int main()
```

```
{
        cout << "\nEnter a string:\n";
        cin.get(str, MAX, '$');          //terminate with $
        cout << "You entered:\n" << str << endl;
        return 0;
}
```
Remember, you must still press Enter after typing the '$' character.

10.6 Copying a String the Hard Way

The following program creates a string constant, str1, and a string variable, str2. It then uses a for loop to copy the string constant to the string variable. The copying is done one character at a time.

```
#include <iostream.h>
#include <cstring.h>                          //for strlen()
int main()
{                                             //initialized string
        char str1[] = "Oh, Captain, my Captain! ";
        const int MAX = 80;                   //size of str2 buffer
        char str2[MAX];                       //empty string
        for(int j=0; j<strlen(str1); j++)     //copy strlen characters
        str2[j] = str1[j];                    // from str1 to str2
        str2[j] = '\0';                       //insert NULL at end
        cout << str2 << endl;                 //display str2
        return 0;
}
```