LESSON SET

7 Arrays

PURPOSE

1. To introduce and allow students to work with arrays
2. To introduce the `typedef` statement
3. To work with and manipulate multidimensional arrays

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.

PRE-LAB READING ASSIGNMENT

One-Dimensional Arrays

So far we have talked about a variable as a single location in the computer's memory. It is possible to have a collection of memory locations, all of which have the same data type, grouped together under one name. Such a collection is called an **array**. Like every variable, an array must be defined so that the computer can "reserve" the appropriate amount of memory. This amount is based upon the type of data to be stored and the number of locations, i.e., size of the array, each of which is given in the definition.

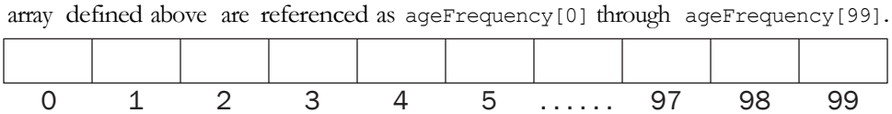
Example: Given a list of ages (from a file or input from the keyboard), find and display the number of people for each age.

The programmer does not know the ages to be read but needs a space for the total number of occurrences of each "legitimate age." Assuming that ages 1, 2, . . . , 100 are possible, the following array definition can be used.

```
const int TOTALYEARS = 100;

int main()
{
   int ageFrequency[TOTALYEARS];     //reserves memory for 100 ints
    :
    return 0;

}
```

Following the rules of variable definition, the data type (integer in this case) is given first, followed by the name of the array (`ageFrequency`), and then the total number of memory locations enclosed in brackets. The number of memory locations must be an integer expression greater than zero and can be given either as a named constant (as shown in the above example) or as a literal constant (an actual number such as 100).

Each element of an array, consisting of a particular memory location within the group, is accessed by giving the name of the array and a position with the array (subscript). In C++ the subscript, sometimes referred to as index, is enclosed in square brackets. The numbering of the subscripts always begins at 0 and ends with one less than the total number of locations. Thus the elements in the `ageFrequency` array defined above are referenced as `ageFrequency[0]` through `ageFrequency[99]`.



If in our example we want ages from 1 to 100, the number of occurrences of age 4 will be placed in subscript 3 since it is the "fourth" location in the array. This odd way of numbering is often confusing to new programmers; however, it quickly becomes routine.[1]

[1] Some students actually add one more location and then ignore location 0, letting 1 be the first location. In the above example such a process would use the following definition: `int agefrequency[101];` and use only the subscripts 1 through 100. Our examples will use location 0. Your instructor will tell you which method to use.

Array Initialization

In our example, `ageFrequency[0]` keeps a count of how many 1s we read in, `ageFrequency[1]` keeps count of how many 2s we read in, etc. Thus, keeping track of how many people of a particular age exist in the data read in requires reading each age and then adding one to the location holding the count for that age. Of course it is important that all the counters start at 0. The following shows the initialization of all the elements of our sample array to 0.

```
for (int pos = 0;  pos < TOTALYEARS;  pos++)
         // pos acts as the array subscript
{
   ageFrequency[pos] = 0;
}
```

A simple for loop will process the entire array, adding one to the subscript each time through the loop. Notice that the subscript (`pos`) starts with 0. Why is the condition `pos < TOTALYEARS` used instead of `pos <= TOTALYEARS`? Remember that the last subscript is one less than the total number of elements in the array. Hence the subscripts of this array go from 0 to 99.

Array Processing

Arrays are generally processed inside loops so that the input/output processing of each element of the array can be performed with minimal statements. Our age frequency program first needs to read in the ages from a file or from the keyboard. For each age read in, the "appropriate" element of the array (the one corresponding to that age) needs to be incremented by one. The following examples show how this can be accomplished:

from a file using `infile` as a logical name

```
infile >> currentAge;
while (infile)

{
  ageFrequency[currentAge-1] =
  ageFrequency[currentAge-1] + 1;
  infile >> currentAge;



}
```

from a keyboard with –99 as sentinel data

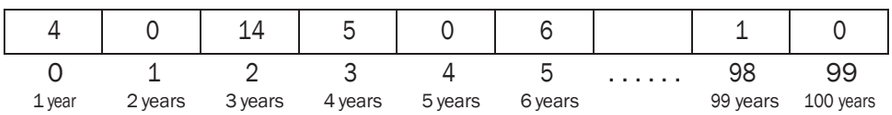
```
cout << "Please input an age from one"
     << "to 100. input -99 to stop"
     << endl;
cin >> currentAge;

while (currentAge != -99)
{
  ageFrequency[currentAge-1] =
  ageFrequency[currentAge-1] + 1;
  cout << "Please input an age from "
       << "one to 100. input -99 to stop"
       << endl;
cin >> currentAge;

}
```

The `while(infile)` statement means that while there is more data in the file `infile`, the loop will continue to process.

To read from a file or from the keyboard we **prime the read**,[2] which means the first value is read in before the test condition is checked to see if the loop

[2] Priming the read for a `while` loop means having an input just before the loop condition (just before the `while`) and having another one as the last statement in the loop.

should be executed. When we read an age, we increment the location in the array that keeps track of the amount of people in the corresponding age group. Since C++ array indices always start with 0, that location will be at the subscript one value less than the age we read in.



Each element of the array contains the number of people of a given age. The data shown here is from a random sample run. In writing the information stored in the array, we want to make sure that only those array elements that have values greater than 0 are output. The following code will do this.

```
for (int ageCounter = 0; ageCounter < TOTALYEARS; ageCounter++)

  if (ageFrequency[ageCounter] > 0)
      cout << "The number of people " << ageCounter + 1 <<" years old is "
           << ageFrequency[ageCounter] << endl;
```

The `for` loop goes from 0 to one less than `TOTALYEARS` (0 to 99). This will test every element of the array. If a given element has a value greater than 0, it will be output. What does outputting `ageCounter + 1` do? It gives the age we are dealing with at any given time, while the value of `ageFrequency[ageCounter]` gives the number of people in that age group.

The complete age frequency program will be given as one of the lab assignments in Lab 7.4.

Arrays as Arguments

Arrays can be passed as arguments (parameters) to functions. Although variables can be passed by value or reference, arrays are always **passed by pointer**, which is similar to pass by reference, since it is not efficient to make a "copy" of all elements of the array. Pass by pointer is discussed further in Lesson Set 9. This means that arrays, like pass by reference parameters, can be altered by the calling function. However, they NEVER have the & symbol between the data type and name, as pass by reference parameters do. Sample Program 7.1 illustrates how arrays are passed as arguments to functions.

Sample Program 7.1:

```
//  The grade average program
//  This program illustrates how one-dimensional arrays are used and how
//  they are passed as arguments to functions.  It contains two functions.
//  The first function is called to allow the user to input a set of grades and
//  store them in an array.  The second function is called to find the average
//  grade.

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

```
const int TOTALGRADES = 50;      // TOTALGRADES is the maximum size of the array

// function prototypes

void getData(int array[], int& sizeOfArray);
  // the procedure that will read values into the array

float findAverage(const int array[], int sizeOfArray);
 // the procedure that will find the average of values
 // stored in an array.  The word const in front of the
 // data type of the array prevents the function from
 // altering the array

int main()
{
      int grades[TOTALGRADES];         // defines an array that holds up to 50 ints
      int numberOfGrades = 0;          // the number of grades read in
      float average;                   // the average of all grades read in

      getData(grades, numberOfGrades); // getData is called to read the grades into
                                       // the array and store how many grades there
                                       // are in numberOfGrades

      average = findAverage(grades, numberOfGrades);

      cout << endl << "The average of the  " << numberOfGrades
           << " grades read in is " << average << "." << endl << endl;

      return 0;
}



//*********************************************************************
//                   getData
//
// task:     This function inputs and stores data in the grades array.
// data in:  none (the parameters contain no information needed by the
//                 getData function)
// data out: an array containing grades and the number of grades
//*********************************************************************

void getData(int array[], int& sizeOfArray)
{
      int pos = 0;                 // array index which starts at 0
      int grade;                   // holds each individual grade read in

      cout << "Please input a grade or type -99 to stop: " << endl;
      cin >> grade;
```

continues

```
      while (grade != -99)
      {
            array[pos] = grade;     // store grade read in to next array location
            pos ++;                 // increment array index

            cout << "Please input a grade or type -99 to stop: " << endl;
            cin >> grade;
      }

      sizeOfArray = pos;            // upon exiting the loop, pos holds the
                                    // number of grades read in, which is sent
                                    // back to the calling function
}

//*************************************************************************
//                              findAverage
//
// task:          This function finds and returns the average of the values
//
// data in:       the array containing grades and the array size
// data returned: the average of the grades contained in that array
//*************************************************************************

float findAverage (const int array[], int sizeOfArray)
{
      int sum = 0;              // holds the sum of all grades in the array

      for (int pos = 0;  pos < sizeOfArray;  pos++)
      {
           sum = sum + array[pos];
          // add grade in array position pos to sum
      }

    return float(sum)/sizeOfArray;
}
```

Notice that a set of empty brackets [ ] follows the parameter of an array which indicates that the data type of this parameter is in fact an array. Notice also that no brackets appear in the call to the functions that receive the array.

Since arrays in C++ are passed by pointer, which is similar to pass by reference, it allows the original array to be altered, even though no `&` is used to designate this. The `getData` function is thus able to store new values into the array. There may be times when we do not want the function to alter the values of the array. Inserting the word **`const`** before the data type on the formal parameter list prevents the function from altering the array even though it is passed by pointer. This is why in the preceding sample program the `findAverage` function and header had the word `const` in front of the data type of the array.

```
float findAverage (const int array[], int sizeOfArray);  // prototype
float findAverage (const int array[], int sizeOfArray)   // function header
```

The variable `numberOfGrades` contains the number of elements in the array to be processed. In most cases not every element of the array is used, which means the size of the array given in its definition and the number of actual elements used are rarely the same. For that reason we often pass the actual number of ele- ments used in the array as a parameter to a procedure that uses the array. The variable `numberOfGrades` is explicitly passed by reference (by using &) to the `getData` function where its corresponding formal parameter is called `sizeOfArray`.

Prototypes can be written without named parameters. Function headers must include named parameters.

```
float findAverage (const int [], int);  // prototype without named parameters
```

The use of brackets in function prototypes and headings can be avoided by declaring a programmer defined data type. This is done in the global section with a **`typedef`** statement.

Example: `typedef int GradeType[50];`

This declares a data type, called `GradeType`, that is an array containing 50 integer memory locations. Since `GradeType` is a data type, it can be used in defining variables. The following defines `grades` as an integer array with 50 elements.

```
GradeType grades;
```

It has become a standard practice (although not a requirement) to use an uppercase letter to begin the name of a data type. It is also helpful to include the word "type" in the name to indicate that it is a data type and not a variable.

Sample Program 7.2 shows the revised code (in bold) of Sample Program 7.1 using `typedef`.

Sample Program 7.2:

```
//  Grade average program
//  This program illustrates how one-dimensional arrays are used and how
//  they are passed as arguments to functions.  It contains two functions.
//  The first function is called to input a set of grades and store them
//  in an array.  The second function is called to find the average grade.

#include <iostream>
using namespace std;


const int TOTALGRADES = 50;            // maximum size of the array

// function prototypes

typedef int GradeType[TOTALGRADES];   // declaration of an integer array data type
                                      // called GradeType
```

continues

```
void getData(GradeType array, int& sizeOfArray);
 // the procedure that will read values into the array

float findAverage(const GradeType array, int sizeOfArray);
 // the procedure that will find the average of values
 // stored in an array.  The word const in front of the
 // data type of the array prevents the function from
 // altering the array

int main()
{
      GradeType grades;              // defines an array that holds up to 50 ints
      int numberOfGrades = 0;        // the number of grades read in
      float average;                 // the average of all grades read in

      getData(grades, numberOfGrades);// getData is called to read the grades into
                                     // the array and store how many grades there
                                     // are in numberOfGrades

      average = findAverage(grades, numberOfGrades);

      cout << endl << "The average of the  " << numberOfGrade
           << " grades read in is " << average << "." << endl << endl;

      return 0;
}

//*********************************************************************
//                   getData
//
// task:     This function inputs and stores data in the grades array.
// data in:  none
// data out: an array containing grades and the number of grades
//*********************************************************************

void getData(GradeType array, int& sizeOfArray)
{
      int pos = 0;                  // array index which starts at 0
      int grade;                    // holds each individual grade read in

      cout << "Please input a grade or type -99 to stop: " << endl;
      cin >> grade;

      while (grade != -99)
      {
            array[pos] = grade;      // store grade read in to next array location
            pos ++;                  // increment array index

            cout << "Please input a grade or type -99 to stop: " << endl;
            cin >> grade;
      }
```

```
      sizeOfArray = pos;     // upon exiting the loop, pos holds the
                             // number of grades read in, which is sent
                             // back to the calling function
}

//*************************************************************************
//                              findAverage
//
// task:          This function finds and returns the average of the values
//
// data in:       the array containing grades and the array size
// data returned: the average of the grades contained in that array
//*************************************************************************

float findAverage (const GradeType array, int sizeOfArray)
{
      int sum = 0;           // holds the sum of all grades in the array

      for (int pos = 0;  pos < sizeOfArray;  pos++)
      {
           sum = sum + array[pos];
           // add grade in array position pos to sum
      }

      return float(sum)/sizeOfArray;
}
```

This method of using `typedef` to eliminate brackets in function prototypes and headings is especially useful for multi-dimensional arrays such as those introduced in the next section.

Two-Dimensional Arrays

Data is often contained in a table of rows and columns that can be implemented with a two-dimensional array. Suppose we want to read data representing profits (in thousands) for a particular year and quarter.

Quarter 1	Quarter 2	Quarter 3	Quarter 4
72	80	10	100
82	90	43	42
10	87	48	53

This can be done using a **two-dimensional array**.

Example:

```
const NO_OF_ROWS = 3;
const NO_OF_COLS = 4;

typedef float ProfitType[NO_OF_ROWS][NO_OF_COLS]; //declares a new data type
                                                  //which is a 2 dimensional
                                                  //array of floats
```

continues

```
int main()
{
    ProfitType profit;         // defines profit as a 2 dimensional array

    for (int row_pos = 0; row_pos < NO_OF_ROWS; row_pos++)
      for (int col_pos = 0; col_pos < NO_OF_COLS;  col_pos++)
      {
          cout << "Please input a profit" << endl;
          cin  >> profit[row_pos][col_pos];
      }
   return 0;
}
```

A two dimensional array normally uses two loops (one nested inside the other) to read, process, or output data.

How many times will the code above ask for a profit? It processes the inner loop `NO_OF_ROWS * NO_OF_COLS` times, which is 12 times in this case.

Multi-Dimensional Arrays

C++ arrays can have any number of dimensions (although more than three is rarely used). To input, process or output every item in an *n*-dimensional array, you need *n* nested loops.

Arrays of Strings

Any variable defined as **`char`** holds only one character. To hold more than one character in a single variable, that variable needs to be an array of characters. A string (a group of characters that usually form meaningful names or words) is really just an array of characters. A complete lesson on characters and strings is given in Lesson Set 10.

PRE-LAB WRITING ASSIGNMENT

Fill-in-the-Blank Questions

1. The first subscript of every array in C++ is _____and the last is _____less than the total number of locations in the array.
2. The amount of memory allocated to an array is based on the _____and the _____of locations or size of the array.
3. Array initialization and processing is usually done inside a _____.
4. The _____ statement can be used to declare an array type and is often used for multidimensional array declarations so that when passing arrays as parameters, brackets do not have to be used.
5. Multi-dimensional arrays are usually processed within _____ loops.
6. Arrays used as arguments are always passed by _____.

7. In passing an array as a parameter to a function that processes it, it is often necessary to pass a parameter that holds the _____ of _____ used in the array.

8. A string is an array of _____.

9. Upon exiting a loop that reads values into an array, the variable used as a(n) _____ to the array will contain the size of that array.

10. An *n*-dimensional array will be processed within _____ nested loops when accessing all members of the array.

LESSON 7A

LAB 7.1 Working with One-Dimensional Arrays

Retrieve program `testscore.cpp` from the Lab 7 folder. The code is as follows:

```
// This program will read in a group of test scores (positive integers from 1 to 100)
// from the keyboard and then calculate and output the average score
// as well as the highest and lowest score. There will be a maximum of 100 scores.

// PLACE YOUR NAME HERE

#include <iostream>
using namespace std;

typedef int GradeType[100];                // declares a new data type:
                                           // an integer array of 100 elements


float findAverage (const GradeType, int);  // finds average of all grades
int   findHighest (const GradeType, int);  // finds highest of all grades
int   findLowest  (const GradeType, int);  // finds lowest of all grades

int main()

{
      GradeType grades;                     // the array holding the grades.
      int numberOfGrades;                   // the number of grades read.
      int pos;                              // index to the array.

      float avgOfGrades;                    // contains the average of the grades.
      int highestGrade;                     // contains the highest grade.
      int lowestGrade;                      // contains the lowest grade.

      // Read in the values into the array

      pos = 0;
      cout << "Please input a grade from 1 to 100, (or -99 to stop)" << endl;
```

continues