Chapter 7

Arrays

Prof. Chien-Nan (Jimmy) Liu
Dept. of Electrical Engineering
National Chiao-Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nctu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

- *7.1 Introduction to Arrays*
- 7.2 Arrays in Functions
- 7.3 Programming with Arrays
- 7.4 Multidimensional Arrays

Introduction to Arrays

- An array is used to process a collection of data of the same type
  - Examples: A list of names
A list of temperatures
- Why do we need arrays?
  - Imagine keeping track of 5 test scores, or 100, or 1000 in memory
    - How would you name all the variables?
    - How would you process each of the variables?

Declaring an Array

- An array, named score, containing five variables of type *int* can be declared as

```
int score[ 5 ];
```

- This is like declaring 5 variables of type int:

  score[0], score[1], … , score[4]
- The value in brackets is called a subscript or an index
  - The index starts from 0, not 1 …
- The variables making up the array are referred to as
  - Indexed variables or elements of the array
- The number of indexed variables in an array is the size of the array
  - The largest index is one less than the size
  - The first index value is zero

Using [ ] With Arrays

- In an array declaration, [ ]'s enclose the size of the array
  - Ex: for an array of 5 integers → int score [5];
- When referring to one indexed variable, the [ ]'s enclose a number identifying the indexed variable
  - Ex: score[3] is one of the indexed variables
  - The value in the [ ]'s can be any expression that evaluates to one of the integers 0 to (size -1)
- To assign a value to an indexed variable, use the assignment operator:

```
int n = 2;
score[n + 1] = 99;
```

→ variable score[3] is assigned 99

Loops And Arrays

- for-loops are commonly used to step through arrays

First index is 0

Last index is (size – 1)

  - Example:

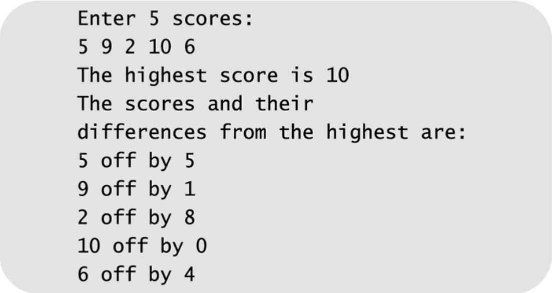
```
for (i = 0; i < 5; i++)
 {
       cout << score[i] << " off by "
        << (max – score[i]) << endl;
 }
```

  could display the difference between each score and the maximum score stored in an array
- Enumeration can help to think about the behavior
  - i = 0 → cout << score[0] << " off by " << max – score[0]
  - i = 1 → cout << score[1] << " off by " << max – score[1]
  - ……

Code Example of Loop + Array

```
//Reads in 5 scores and shows how much each
//score differs from the highest score.
#include <iostream>
int main( )
{
   using namespace std;
   int i, score[5], max;

   cout << "Enter 5 scores:\n";
   cin >> score[0];
   max = score[0];
   for (i = 1; i < 5; i++)
   {
      cin >> score[i];
      if (score[i] > max)
         max = score[i];
      //max is the largest of the values score[0],
..., score[i].
   }
   cout << "The highest score is " << max << endl
        << "The scores and their\n"
        << "differences from the highest are:\n";
   for (i = 0; i < 5; i++)
      cout << score[i] << " off by "
           << (max - score[i]) << endl;

   return 0;
}
```

Sample Dialogue



```
Enter 5 scores:
5 9 2 10 6
The highest score is 10
The scores and their
differences from the highest are:
5 off by 5
9 off by 1
2 off by 8
10 off by 0
6 off by 4
```

Constants and Arrays

- Use constants to declare the size of an array
  - Using a constant allows your code to be easily altered for use on a smaller or larger set of data
    - Example:

```
const int  NUMBER_OF_STUDENTS = 50;
int score[NUMBER_OF_STUDENTS];
…
for ( i = 0; i < NUMBER_OF_STUDENTS; i++)
   cout << score[i] << " off by "
        << (max – score[i]) << endl;
```

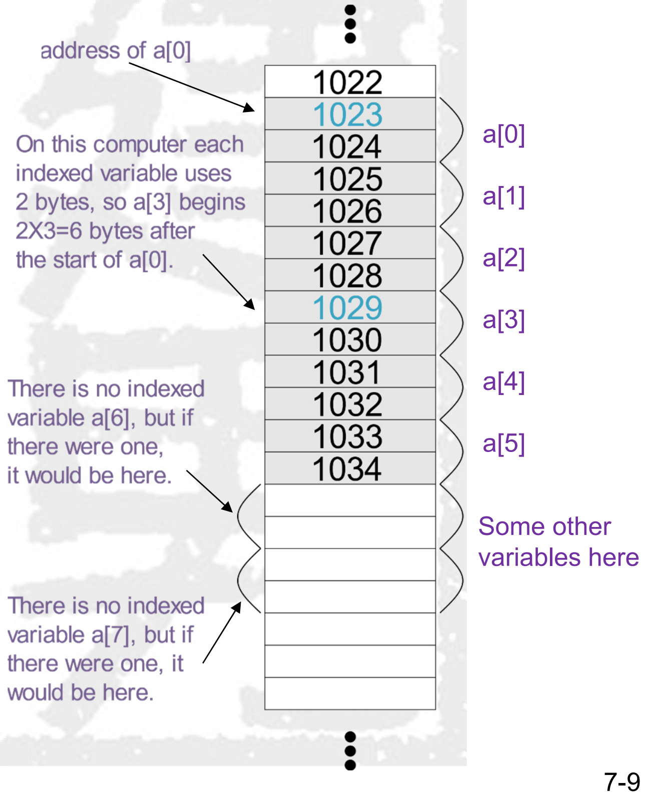
- Some compilers do not allow the use of a variable to declare the size of an array
  - Example:

```
cout << "Enter number of students: ";
cin >> number;
int score[number];
```

→ Not sure how many seats should be reserved at compile time

Arrays and Memory

- Declaring the array int a[6]
  - Reserves memory for six variables of type *int*
  - The variables are stored one after another (consecutive locations)
  - Only the address of a[0] is remembered
  - To determine the address of a[3]
    - Start at a[0]
    - Count the memory for three integers
    - Past enough memory to find a[3]



address of a[0]

On this computer each indexed variable uses 2 bytes, so a[3] begins 2X3=6 bytes after the start of a[0].

There is no indexed variable a[6], but if there were one, it would be here.

There is no indexed variable a[7], but if there were one, it would be here.

Some other variables here

Out of Range Problems

- A common error is using a nonexistent index
  - Index values for int a[6] are the values 0 through 5, not 1 to 6
  - An index value not allowed by the array declaration is out of range, ex: using a[7] ??
- Using an out of range index value does not produce an error message!!
  - However, this address could be where some other variable is stored
  - May cause some unpredictable errors!!

Initializing Arrays

- To initialize an array when it is declared
  - The values for the indexed variables are enclosed in braces and separated by commas
  - Example: int children[3] = { 2, 12, 1 };
    Is equivalent to:

```
int children[3];
children[0] = 2;
children[1] = 12;
children[2] = 1;
```

- If the array is not initialized, the variables in the array may be zero or an unpredictable value
  - Dangerous to use unpredictable values for calculation

Default Initial Values

- If too few values are listed in an initialization statement
  - The listed values are used to initialize the first of the indexed variables
  - The remaining indexed variables are initialized to a zero of the base type
  - Example:

```
int a[10] = {5, 5};
```

→ initializes a[0] and a[1] to 5 and
a[2] through a[9] to 0

```
int a[10] = {0};
```

→ initializes all elements to 0

Overview

- *7.1 Introduction to Arrays*
- *7.2 Arrays in Functions*
- 7.3 Programming with Arrays
- 7.4 Multidimensional Arrays

Arrays in Functions

- Indexed variables can be arguments to functions
  - Example: If a program contains these declarations:

```
int i, n, a[10];
void myFunction(int n);
```

a single integer

  - Variables a[0] through a[9] are of type *int*, making these calls legal:

```
myFunction( a[ 0 ] );
myFunction( a[ 3 ] );
myFunction( a[ i ]  );
```

- Just like typical variables, indexed variables are passed-by-value

Arrays as Function Arguments

- A formal parameter can be for an entire array
  - Such a parameter is called an array parameter
- Passing the array name as an argument to represent the whole array elements
  - Not a call-by-value nor a call-by-reference parameter
  - Just pass the address of the first array element
- Array parameters behave much like call-by-reference parameters
  - The values of the indexed variables can be changed by the function
  - Avoid duplicating whole array data

Function Calls With Arrays

- An array parameter is indicated using empty brackets in the parameter list such as

```
void fillUp(int a[ ], int size);
```

- If function fillUp is declared in this way, and array score is declared this way:

```
int score[5], numberOfScores;
```

- fillUp is called in this way:

```
fillUp(score, numberOfScores);
```

- However, functions cannot return arrays
  - In Chap. 9, we will learn how to return a pointer to an array

Code: Passing Array to Function

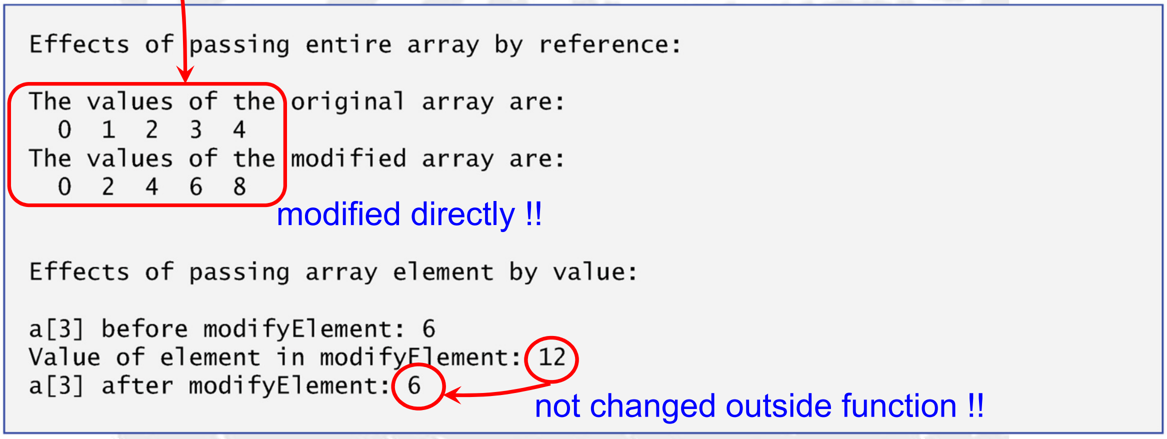
```
#include <iostream>
#include <iomanip>
using namespace std;

void modifyArray( int [], int );
void modifyElement( int );

int main()
{
  const int arraySize = 5;  // size of array a
  int a[arraySize] = {0,1,2,3,4};  // initialize array

  cout << "Effects of passing entire array by reference:"
      << "\n\nThe values of the original array are:\n";

  // output original array elements
  for (int i=0; i < arraySize; i++)
    cout << setw(3) << a[i];
  cout << endl;

  // pass array a by reference
  modifyArray(a, arraySize);
  cout << "The values of the modified array are:\n";
```

```
  // output modified array elements
  for (int j=0; j < arraySize; j++)
    cout << setw(3) << a[j];

  cout << "\n\n\nEffects of passing element by value:"
       << "\n\na[3] before modifyElement: "
       << a[3] << endl;

  // pass array element a [3] by value
  modifyElement(a[3]);
  cout << "a[3] after modifyElement: “ << a[3] << endl;
}  // end main

void modifyArray(int b[], int sizeOfArray)
{
  // multiply each array element by 2
  for (int k=0; k < sizeOf Array; k++)
    b[k] *= 2;
}  // end function modifyArray

void modifyElement(int e)
{
  // multiply parameter by 2
  cout << "Value of element in modifyElement: “
       << (e *= 2) << endl;
}  // end function modifyElement
```

Pass-by-Ref vs Pass-by-Value

```
void modifyArray(int b[], int sizeOfArray)
{
  // multiply each array element by 2
  for (int k=0; k < sizeOf Array; k++)
    b[k] *= 2;
}  // end function modifyArray
```

```
void modifyElement(int e)
{
  // multiply parameter by 2
  cout << "Value of element in modifyElement: "
       << (e *= 2) << endl;
}  // end function modifyElement
```



```
Effects of passing entire array by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8

Effects of passing array element by value:

a[3] before modifyElement: 6
Value of element in modifyElement: 12
a[3] after modifyElement: 6
```

modified directly !!

not changed outside function !!

Array Parameter Considerations

- What does a function know about an array argument?
  - The base type
  - The address of the first indexed variable
- Because a function does not know the size of an array argument…
  - The programmer should include a formal parameter that specifies the size of the array
  - The function can process arrays of various sizes
    - Ex: fillUp(score, 5); fillUp(time, 10); …

const Modifier

- Array parameters allow a function to change the values stored in the array argument
  - Behave like call-by-reference parameters
- If a function should not change the values of the array argument, use the modifier **const**
  - It is called a constant array parameter
  - Ex: void showTheWorld(const int a[ ], int size);
- *const* is used in both the function declaration and definition to modify the array parameter
- The compiler will issue an error if you want to change the values stored in the constant array

const Parameters Example

```
double computeAverage(int a[ ], int size);

void showDifference(const int a[ ], int size)
{
      double average = computeAverage(a, size);
       …
}
```

- *computeAverage* has no constant array parameter
- This code generates an error message because *computeAverage* could change the array parameter
- In this case, the called function must use a constant array parameter as a placeholder for the array

Case Study: Production Graph

- Problem Definition:
  - We are writing a program for the Apex Plastic Spoon Company
  - The program will display a bar graph showing the production of each of four plants for a week
  - Each plant has separate records for each department
  - Input is entered plant by plant
  - Output shows one asterisk for each 1000 units, and production is rounded to the nearest 1,000 units

Analysis of The Problem

- Use an array named *production* to hold total production of each plant
  - Production for plant n is stored in production[n-1]
- Program must scale production to nearest 1,000 units to display asterisks in the bar
- The entire array will be an argument for the functions we write to perform the subtasks
  - Also include a formal parameter for the size
  - The size of the array is equal to the number of plants
  - We will use a constant for the number of plants

Production Graph Sub-Tasks

- Analysis leads to the following sub-tasks
  - inputData: Read input for each plant
    Set production [plantNumber -1]
    to the total production for plant number n
  - scale: For each plant, change
    production[plantNumber]
    to the correct number of asterisks
  - graph: Output the bar graph

Code for Production Graph: Main

```
#include <iostream>
const int NUMBER_OF_PLANTS = 4;

void inputData(int a[], int lastPlantNumber);
void scale(int a[], int size);
void graph(const int asterisk_count[], int lastPlantNumber);

int main( )
{
   using namespace std;
   int production[NUMBER_OF_PLANTS];

   cout << "This program displays a graph showing\n"
       << "production for each plant in the company.\n";

   inputData(production, NUMBER_OF_PLANTS);
   scale(production, NUMBER_OF_PLANTS);
   graph(production, NUMBER_OF_PLANTS);

   return 0;
}
```

Algorithm Design: inputData

- Read all departments' data for each plant and add them to produce a plant's total
  - Algorithm for inputData:

```
for plantNumber is 1, 2, …,
      lastPlantNumber
do the following
     Read the data for each plant
     Sum the numbers
     Set production[plantNumber–1]
     to the total
```

```
void inputData(int a [ ], int lastPlantNumber)
{
     using namespace std;

     for (int plantNumber = 1;
          plantNumber <= lastPlantNumber;
         plantNumber++)
     {
          cout << endl;
         << "Enter production for plant"
         << plantNumber << endl;
          getTotal( a[plantNumber -1] );
      }
 }
```

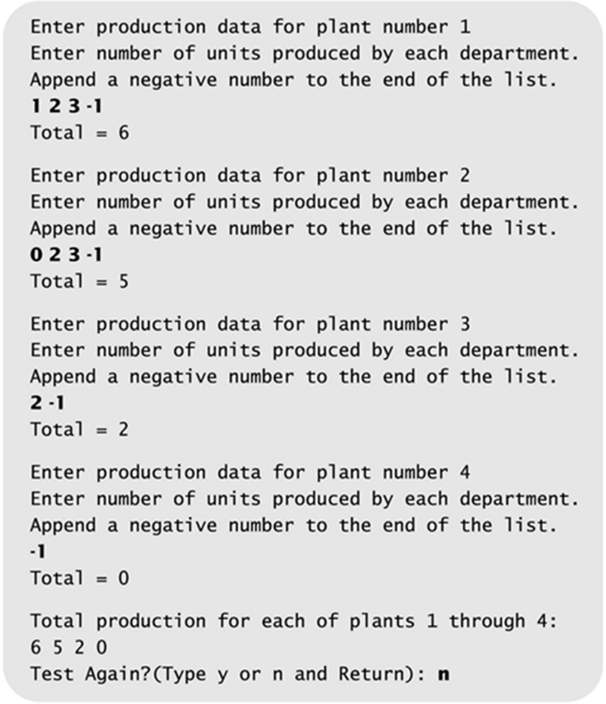
Testing inputData

- Each function should be tested in a program in which it is the only untested function
- *inputData* calls *getTotal*, *getTotal* is tested first
- Once tested, *getTotal* can be used to test inputData
- Remember that inputData should be tested
  - With a plant that contains no production figures
  - With a plant having only one production figure
  - With a plant having more than one figure
  - With zero and non-zero production figures

Code for getTotal

```
//Uses iostream:
void getTotal(int& sum)
{
   using namespace std;
   cout << "Enter number of units
produced by each department.\n"
       << "Append a negative number to
the end of the list.\n";

   sum = 0;
   int next;
   cin >> next;
   while (next >= 0)
   {
      sum = sum + next;
      cin >> next;
   }
   cout << "Total = " << sum << endl;
}
```

Sample Dialogue



```
Enter production data for plant number 1
Enter number of units produced by each department.
Append a negative number to the end of the list.
1 2 3 -1
Total = 6

Enter production data for plant number 2
Enter number of units produced by each department.
Append a negative number to the end of the list.
0 2 3 -1
Total = 5

Enter production data for plant number 3
Enter number of units produced by each department.
Append a negative number to the end of the list.
2 -1
Total = 2

Enter production data for plant number 4
Enter number of units produced by each department.
Append a negative number to the end of the list.
-1
Total = 0

Total production for each of plants 1 through 4:
6 5 2 0
Test Again?(Type y or n and Return): n
```

Algorithm for scale

- scale changes the value of the indexed variable to show the whole number of asterisks to print
  - Scale is called using
    scale (production, NUMBER_OF_PLANTS);
  - For each index < size, divide the value of a[index] by 1,000 and round the result to the nearest integer
- The code for scale uses a function named *round* that must be defined as well

  -
    ```
    void scale(int a[ ], int size)
    {
        for (int index = 0; index < size; index++)
           a[index] = round (a[index] / 1000.0);
    }
    ```

Why not 1000?

Function floor

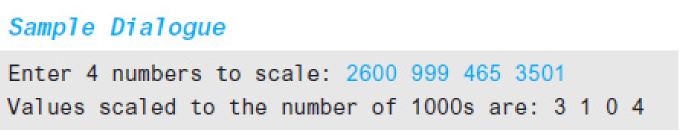
- Function round, called by scale, uses the floor function from the <cmath> library
  - The floor function returns the first whole number less than its argument:
    floor (3.4) returns 3
    floor (3.9) returns 3
  - Adding 0.5 to the argument for floor is how round performs its task
    floor (3.4 + 0.5) returns 3
    floor (3.9 + 0.5) returns 4

Testing scale Function

- Scale should be tested for zero, round up, and round down

```
int main( )
{
   using namespace std;
   int someArray[4], index;

   cout << "Enter 4 numbers to scale: ";
   for (index = 0; index < 4; index++)
      cin >> someArray[index];

   scale(someArray, 4);

   cout << "Values scaled to the number "
         << " of 1000s are: ";
   for (index = 0; index < 4; index++)
      cout << someArray[index] << " ";
   cout << endl;

   return 0;
}
```

```
void scale(int a[], int size)
{
   for (int index = 0; index < size; index++)
      a[index] = roundNum(a[index]/1000.0);
}

//Uses cmath:
int roundNum(double number)
{
   using namespace std;
   return static_cast<int>(floor(number + 0.5));
}
```



Sample Dialogue

```
Enter 4 numbers to scale: 2600 999 465 3501
Values scaled to the number of 1000s are: 3 1 0 4
```

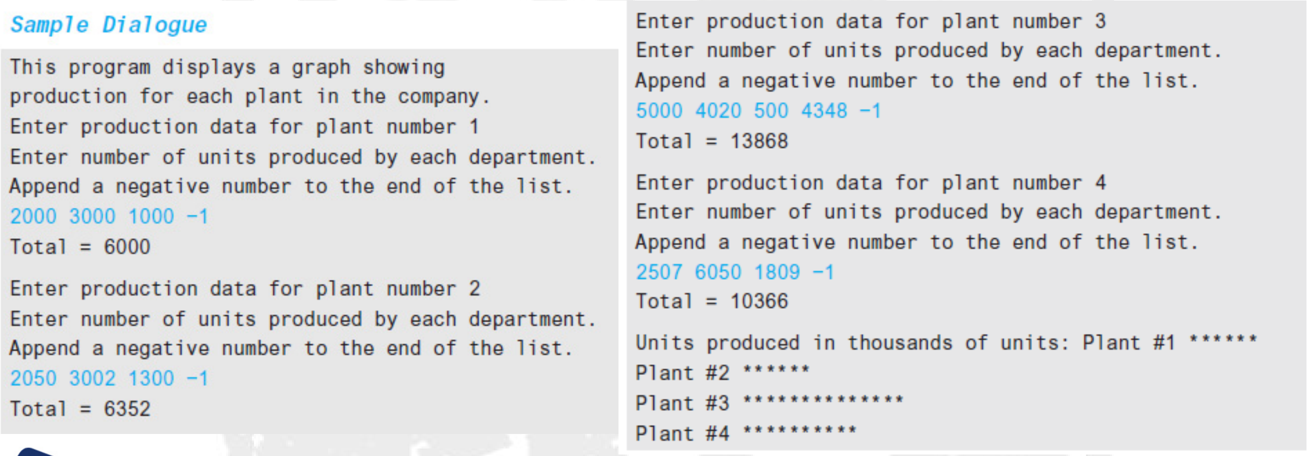
Function graph

- The design of graph is quite straightforward
  - Use a for loop to print the desired number of '*'

```
void graph(const int asteriskCount[], int lastPlantNumber) {
   using namespace std;
   cout << "\nUnits produced in thousands of units:\n";
   for (int plant_number = 1; plant_number <= lastPlantNumber; plant_number++)
   {
      cout << "Plant #" << plant_number << " ";
      printAsterisks(asteriskCount[plant_number - 1]);
      cout << endl;
   }
}

void printAsterisks(int n) {
   using namespace std;
   for (int count = 1; count <= n; count++)
      cout << "*";
}
```

Final Results of Production Graph

- The complete program can be found in the text book at page 441-443
  - Here shows some testing results



Sample Dialogue

```
This program displays a graph showing
production for each plant in the company.
Enter production data for plant number 1
Enter number of units produced by each department.
Append a negative number to the end of the list.
2000 3000 1000 -1
Total = 6000

Enter production data for plant number 2
Enter number of units produced by each department.
Append a negative number to the end of the list.
2050 3002 1300 -1
Total = 6352
```

```
Enter production data for plant number 3
Enter number of units produced by each department.
Append a negative number to the end of the list.
5000 4020 500 4348 -1
Total = 13868

Enter production data for plant number 4
Enter number of units produced by each department.
Append a negative number to the end of the list.
2507 6050 1809 -1
Total = 10366

Units produced in thousands of units: Plant #1 ******
Plant #2 ******
Plant #3 **************
Plant #4 **********
```

Overview

- *7.1 Introduction to Arrays*
- *7.2 Arrays in Functions*
- *7.3 Programming with Arrays*
- 7.4 Multidimensional Arrays

Programming With Arrays

- The size needed for an array is changeable
  - Often varies from one run of a program to another
  - Is often not known when the program is written
- However, array declaration requires a fixed size
- A common solution to the size problem
  - Declare the array size to be the largest that could be needed. Ex: a[100]
  - Use an extra parameter, *number_used*, to ensure that referenced index values are legal
  - This is called a partially filled array in this book …

Code: Partially Filled Array (1/2)

```
#include <iostream>
const int MAX_NUMBER_SCORES = 10;
using namespace std;

int main( ){
   int score[MAX_NUMBER_SCORES], numberUsed;
   cout << "This program reads golf scores and shows\n"
        << "how much each differs from the average.\n";
   cout << "Enter golf scores:\n";
   fillArray(score, MAX_NUMBER_SCORES, numberUsed);
   showDifference(score, numberUsed);
   return 0;
}

void fillArray(int a[], int size, int& numberUsed){
   cout << "Enter up to " << size << " nonnegative whole numbers.\n"
        << "Mark the end of the list with a negative number.\n";
   int next, index = 0;
   cin >> next;
   while ((next >= 0) && (index < size)){
      a[index] = next;
      index++;
      cin >> next;
   }
   numberUsed = index;
}
```

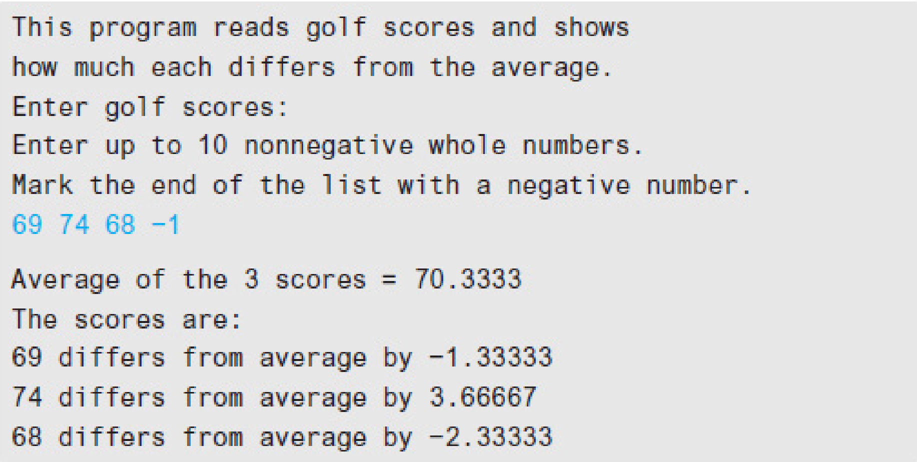
calculate actual number used in the array

Code: Partially Filled Array (2/2)

```
double computeAverage(const int a[], int numberUsed){
    double total = 0;
    for (int index = 0; index < numberUsed; index++)
        total = total + a[index];
    if (numberUsed > 0)
        return (total/numberUsed);
    else
        ……
```

use actual number to traverse the for loop

Sample Dialogue



```
This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
69 74 68 -1

Average of the 3 scores = 70.3333
The scores are:
69 differs from average by -1.33333
74 differs from average by 3.66667
68 differs from average by -2.33333
```

Searching Arrays

- A sequential search is one way to search an array for a given value
  - Look at each element from first to last to see if the target value is equal to any of the array elements
    - Set the tag as TRUE and break searching immediately
  - The index of the target value can be returned to indicate where the value was found in the array
    - The stopped location is the desired index
  - A value of -1 can be returned if the value was not found
    - If the tag is still FALSE after traversing the whole loop

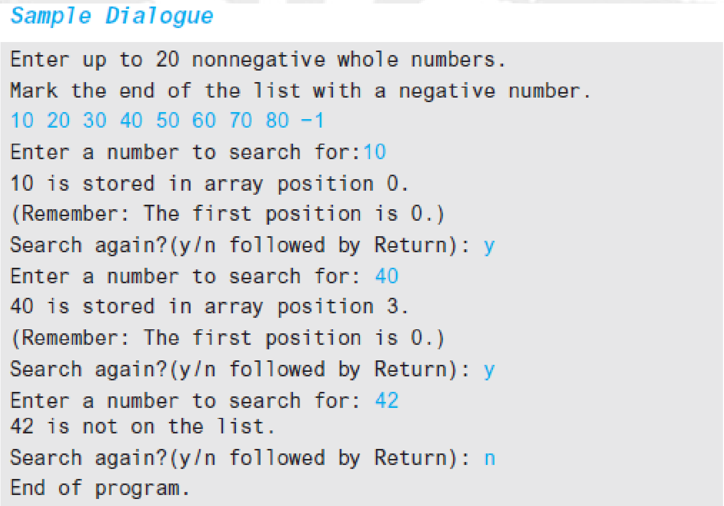
The search Function

- Use a while loop to compare array elements to the target value
- Set the variable *found* of type bool after comparison
  - Initially FALSE. Set to TRUE if found…
- Check the variable *found* at the end to determine the return value

```
int search(const int a[], int numberUsed, int target) {
    int index = 0;
    bool found = false;
    while ((!found) && (index < numberUsed))
        if (target == a[index])   found = true;
        else   index++;
    if (found)   return index;
    else   return -1;
}
```

Test Array Searching

- Use a do-while loop to allow users search for multiple numbers in the array
  - Input ‘n’ to finish searching. Otherwise, search again…
- Test for found and unfound numbers



Sample Dialogue

```
Enter up to 20 nonnegative whole numbers.
Mark the end of the list with a negative number.
10 20 30 40 50 60 70 80 -1
Enter a number to search for:10
10 is stored in array position 0.
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 40
40 is stored in array position 3.
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 42
42 is not on the list.
Search again?(y/n followed by Return): n
End of program.
```

Sorting an Array

- Sorting a list of values is very common
  - Create an alphabetical listing
  - Create a list of values in ascending order
  - Create a list of values in descending order
- When the sort is complete, the elements of the array are ordered such that

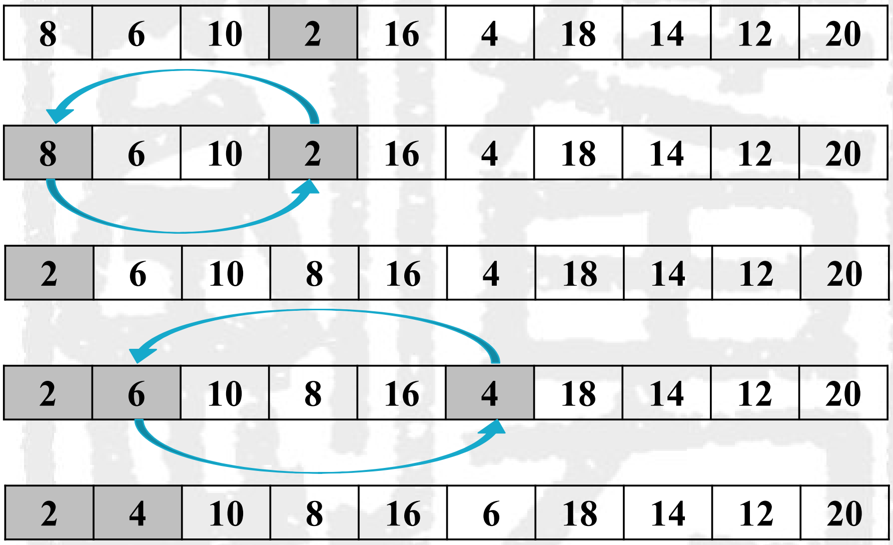
  a[0] < a[1] < … < a [number_used -1]
  - Descending order is also allowed
- Many sorting algorithms exist
  - Some are very efficient
  - Some are easier to understand

Selection Sort Algorithm

- One array is sufficient to do our sorting
  - Search for the smallest value in the array
    → a separate function *indexOfSmallest* will do that
  - Exchange the values at smallest location and a[0]
    → smallest value will be placed at the first place
  - Starting at a[1], find the smallest remaining value swap it with the value currently in a[1]
  - Starting at a[2], repear the process until the array is sorted

Selection Sort Illustrated

Selection Sort



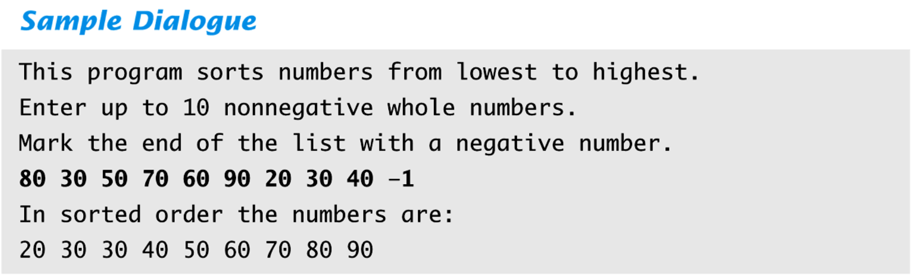
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	10	2	16	4	18	14	12	20
8	6	10	2	16	4	18	14	12	20
2	6	10	8	16	4	18	14	12	20
2	6	10	8	16	4	18	14	12	20
2	4	10	8	16	6	18	14	12	20

Code: Selection Sort (1/2)

```
void sort(int a[], int numberUsed)
{
    int indexOfNextSmallest;
    for (int index = 0; index < numberUsed - 1; index++)
    {
        //Place the correct value in a[index]:
        indexOfNextSmallest = indexOfSmallest(a, index, numberUsed);
        swapValues(a[index], a[indexOfNextSmallest]);
        //a[0] <= a[1] <=...<= a[index] are the smallest of the original array
        //elements. The rest of the elements are in the remaining positions.
    }
}

void swapValues(int &v1, int &v2)
{
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

Code: Selection Sort (2/2)

```
int indexOfSmallest(const int a[], int startIndex, int numberUsed)
{
   int min = a[startIndex],
   indexOfMin = startIndex;
   for (int index = startIndex + 1; index < numberUsed; index++)
      if (a[index] < min)
      {
         min = a[index];
         indexOfMin = index;
         //min is the smallest of a[startIndex] through a[index]
      }

   return indexOfMin;
}
```



Sample Dialogue

```
This program sorts numbers from lowest to highest.
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
80 30 50 70 60 90 20 30 40 -1
In sorted order the numbers are:
20 30 30 40 50 60 70 80 90
```

Bubble Sort Algorithm

- Another simple sorting algorithm is Bubble Sort
- Key idea: bubble the largest value toward the end of the array (ascending or descending?)
    - Swapping consecutive elements only
- In the second run, the algorithm repeats the process but stops at the position to the left of previous location (second largest)
- If there are n elements, repeat “bubble” n-1 times
- Implementation requires nested loops

Bubble Sort: The First Run

- Initial array:
  3, 10, 9, 2, 5
- Compare 3 and 10; no swap since 10 is greater than 3
  3, 10, 9, 2, 5
  △ △
- Compare 10 and 9; swap since 10 is larger than 9
  3, 10, 9, 2, 5
  △ △
- Compare 10 and 2; swap since 10 is larger than 2
  3, 9, 10, 2, 5
  △ △
- Compare 10 and 5; swap since 10 is larger than 5
  3, 9, 2, 10, 5
  △ △
- We have "bubbled" the largest value to the right
  3, 9, 2, 5, 10

Code: Bubble Sort

```
void bubblesort(int arr[], int length){
        // Bubble largest number toward the right
        for (int i = length - 1; i > 0; i--)
                for (int j = 0; j < i; j++)
                        if (arr[j] > arr[j + 1])
                        {
                                // Swap the numbers
                                int temp = arr[j + 1];
                                arr[j + 1] = arr[j];
                                arr[j] = temp;
                        }
}
```

```
Initial array:   3 10 9 2 5 1
Sorted array: 1 2 3 5 9 10
```

Overview

- 7.1 Introduction to Arrays
- 7.2 Arrays in Functions
- 7.3 Programming with Arrays
- *7.4 Multidimensional Arrays*

Multi-Dimensional Arrays

- C++ allows arrays with multiple index values
  - char page [30] [100];
    declares an array of characters named page
    - This array has two index values:
      The first ranges from 0 to 29
      The second ranges from 0 to 99
  - Each index is enclosed in its own brackets
  - Page can be visualized as an array of 30 rows and 100 columns
- C++ supports two dimensions only
  - For more dimensions, use index transformation
  - Ex: 2D transformation
    - x, y: reference addr.
    - real index = y*4 + x

	x=0	x=1	x=2	x=3
y=0	**0**	**1**	**2**	**3**
y=1	**4**	**5**	**6**	**7**
y=2	**8**	**9**	**10**	**11**

Index Values of 2-D Array

- The indexed variables for array page are

page[0][0]	page[0][1]	……	page[0][99]
page[1][0]	page[1][1]	……	page[1][99]
……	……	……	……
page[29][0]	page[29][1]	……	page[29][99]

- Referencing a 2-D array element page[x][y] incorrectly as page[x,y] is a common error !!
  - Actually, C++ evaluates the comma-separated expression (x,y) simply as y (the last expressions)

Initialize Multidimensional Array

```
const int rows = 2, columns = 3;
int main()
{
    int array1[rows][columns] = { {1,2,3}, {4,5,6} };
    int array2[rows][columns] = { 1,2,3,4,5 };
    int array3[rows][columns] = { {1,2}, {4} };

    printArray(array1);
    printArray(array2);
    printArray(array3);
}
```

```
Values in array1 are:
1 2 3
4 5 6
Values in array2 are:
1 2 3
4 5 0
Values in array3 are:
1 2 0
4 0 0
```

If there are not enough initializers for a given row, the remaining elements of that row are set to 0.

Passing Multidimensional Array

- While using one-dimensional array as a formal parameter, the size of array is not needed!

```
void displayLine(const char a[ ], int size);
```

- While using a multi-dimensional array, the column size must be completely specified in the parameter declaration

```
void displayPage(const char page[ ] [100],
                 int sizeDimension1);
```

  - Compiler requires the column size to do the index transformation for you…

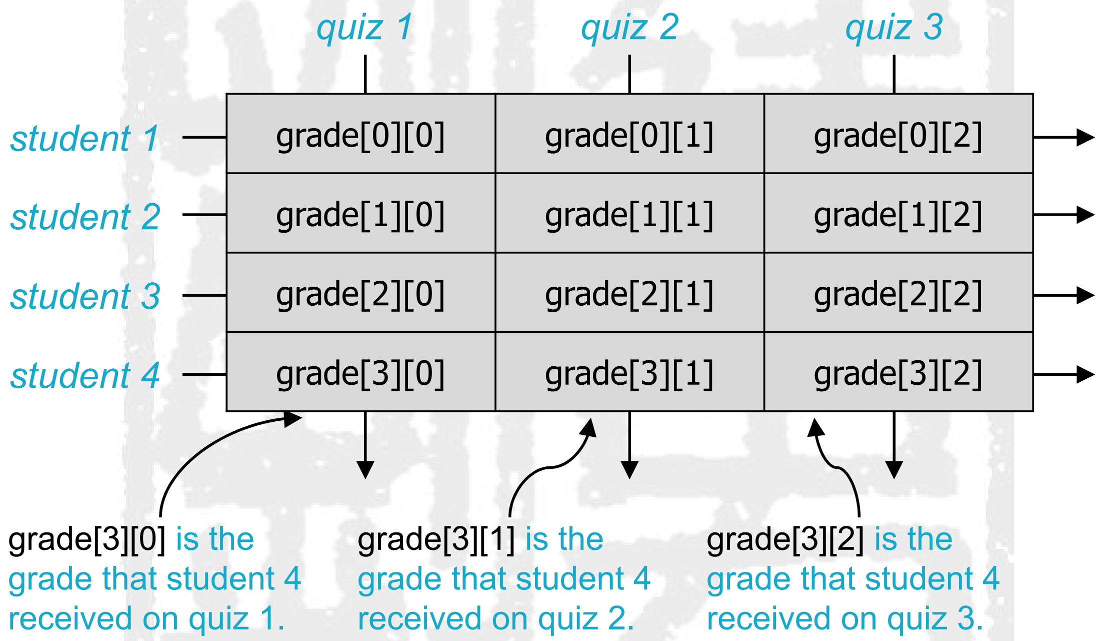
Example: Grading Program

- Grade records for a class can be stored in a two-dimensional array
  - For a class with 4 students and 3 quizzes the array could be declared as

```
int grade[4][3];
```

    - The first array index refers to the number of a student
    - The second array index refers to a quiz number
- Since student and quiz numbers start with one, we subtract one to obtain the correct index
  - No.1, No.2, No.3, … → g[0], g[1], g[2], …

The Two-Dimensional Array



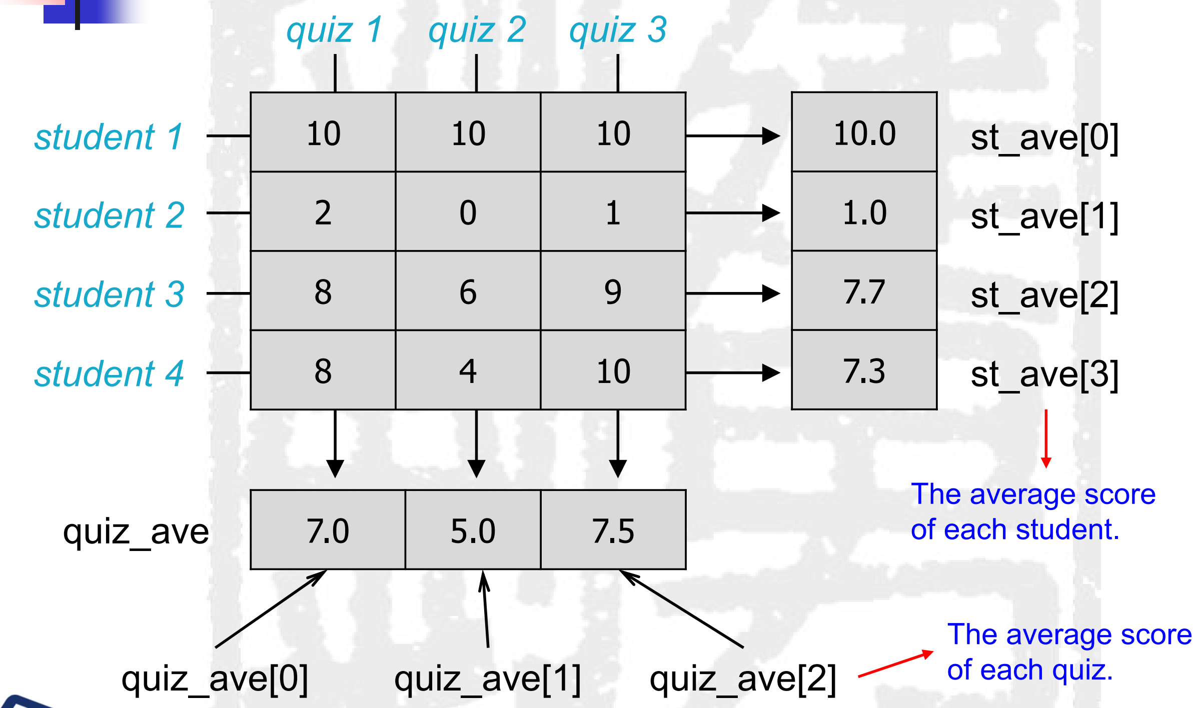
	quiz 1	quiz 2	quiz 3
student 1	grade[0][0]	grade[0][1]	grade[0][2]
student 2	grade[1][0]	grade[1][1]	grade[1][2]
student 3	grade[2][0]	grade[2][1]	grade[2][2]
student 4	grade[3][0]	grade[3][1]	grade[3][2]

grade[3][0] is the grade that student 4 received on quiz 1.

grade[3][1] is the grade that student 4 received on quiz 2.

grade[3][2] is the grade that student 4 received on quiz 3.

Calculate Average Scores



	quiz 1	quiz 2	quiz 3		
student 1	10	10	10	10.0	st_ave[0]
student 2	2	0	1	1.0	st_ave[1]
student 3	8	6	9	7.7	st_ave[2]
student 4	8	4	10	7.3	st_ave[3]
quiz_ave	7.0	5.0	7.5		
	quiz_ave[0]	quiz_ave[1]	quiz_ave[2]		

The average score of each student.

The average score of each quiz.

Code for Average Scores

```
void computeStAve(const int grade[][NUMBER_QUIZZES], double stAve[]){
    for (int stNum = 1; stNum <= NUMBER_STUDENTS; stNum++) {   // per stNum
        double sum = 0;
        for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
            sum = sum + grade[stNum -1][quizNum -1];
        //sum contains the sum of the quiz scores for student number stNum.
        stAve[stNum -1] = sum/NUMBER_QUIZZES;
        //Average for student stNum is the value of stAve[stNum-1]
    }
}
```

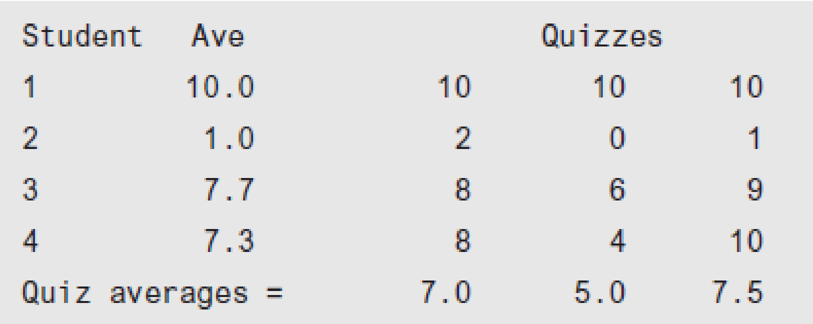
nested for loops to control two indexes

```
void computeQuizAve(const int grade[][NUMBER_QUIZZES], double quizAve[])
{
    for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++) {   // per quiz
        double sum = 0;
        for (int stNum = 1; stNum <= NUMBER_STUDENTS; stNum++)
            sum = sum + grade[stNum - 1][quizNum - 1];
        //sum contains the sum of all student scores on quiz number quizNum.
        quizAve[quizNum - 1] = sum / NUMBER_STUDENTS;
        //Average for quiz quizNum is the value of quizAve[quizNum-1]
    }
}
```

given two indexes, it is used as an integer

Test Data for Grading Program

```
const int NUMBER_STUDENTS=4, NUMBER_QUIZZES=3;
int main( ){
    using namespace std;
    int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];
    double stAve[NUMBER_STUDENTS];
    double quizAve[NUMBER_QUIZZES];
    //The code for filling the array grade goes here, but is not shown.
    computeStAve(grade, stAve);
    computeQuizAve(grade, quizAve);
    display(grade, stAve, quizAve);
    return 0;
}
```



```
Student   Ave              Quizzes
1         10.0        10      10      10
2          1.0         2       0       1
3          7.7         8       6       9
4          7.3         8       4      10
Quiz averages =      7.0     5.0     7.5
```