Programming in C/C++

Chapter 7: Built In Arrays

Kristina Shroyer

Objectives

You should be able to describe:

- One-Dimensional Arrays
- Array Initialization
- Array Elements as Arguments to Functions
- Entire Arrays as Arguments to Functions
- Parallel Arrays
- Two-Dimensional Arrays
- Common Programming Errors

IMPORTANT!!!

- What we are discussing in this chapter are BUILT IN ARRAYS
  - This is different than a fixed array in the new C++ standard
    - The new standard has two types of arrays: this one the built in array and a data type (fixed size array)
  - In this class we are learning built-in arrays – built in arrays are NOT class data types – instead we are working with the raw memory of the computer

- ***For that reason I do NOT want you using the internet or any other source for this material other than the slides or our book (which I know discusses built-in arrays)***

- ***Even if I don't specifically say BUILT in array during the lecture notes – that is what we are talking about***
  - ***Some of the concepts DO apply to arrays in general but to avoid confusion just think of everything as saying BUILT in array***

Arrays Hold Multiple Values

- **Concept:** An array allows you to store and work with multiple values of the ***same*** data type
  - The majority of the variables we have worked with so far are designed to hold only one value at a time
  - Each of the variable definitions below cause only enough memory to be reserved to hold one value of the specified data type

int count;	12314	Enough memory for **1** `int`
float price;	56.981	Enough memory for **1** `float`
char letter;	A	Enough memory for **1** `char`

Arrays

- **An array can hold multiple values of the same data type simultaneously**
  - **All the values are stored in consecutive memory locations**
    - Note an ***built in array*** is not a class data type (a definition of an object) in C++ like it is in Java – so when you use a ***built in array*** in C++ you are actually working with the memory locations and are NOT working with an object (instance of a class data type)
  - **Think of an array as a list of related values**
    - All items in list have ***same data type***
    - All list members stored using ***single group name***
      - This group name is referred to as the **array name**
    - The members are stored in ***consecutive memory locations***

- An array is a powerful programming device for handling large quantities of ***related data of the same type***
  - Allows for:
    - Storage of elements
      - Can use it to store elements in a particular order
    - Retrieval/Access of elements
    - Searching through elements
    - Changing Elements

Arrays

- **Visual Representation of an Array**

- **When Declaring a built-in array (setting aside memory for it) you need to specify three things:**
  - **The Name of the Array**
  - **The Number of Elements in the Array**
    - An Array can be declared to hold as many elements as you need
    - The Array below would hold 11 elements
    - Once you declare the number of elements in an array it cannot change
      - Array size is fixed
  - **The Data Type of the Elements in the Array**
    - The Elements in an Array must all be of the same data type

Arrays

- A built-in array declaration statement provides:
  - The array name
  - The data type of array items
  - The number of items in array

- Syntax

```
dataType arrayName[numberOfItems];
```

- Example:

```
int grades[11];
```

- The name of this built-in array is **`grades`**
- The number inside the brackets is the array's ***size declarator***
  - `dataType arrayName[size declarator];`
  - The ***size declarator*** indicates the number of elements or values the built-in array can hold
- The data type of the array is **`int`** meaning all of the values in the array are of the integer data type

`grades` contains the address of the first element of the array

`grades` array: enough memory to hold 11 **`int`** values

Array element (an `int`) – the first of 11 in this array

Built-in Arrays

- A built-in array's **size declarator:**
  - ***Must be a constant integer expression with a value greater than zero***
    - ***This means it must be a const variable or a literal***
    - ***It can NOT be a regular variable***
  - It can be a literal (previous example) or a named constant as shown here:

```
const int numGrades = 11;
int grades[numGrades];
```

  - Common programming practice requires defining number of built-in array items as a constant before declaring the array
    - This constant is extremely useful later for processing all the items in a built-in array.

`grades` array: enough memory to hold 11 `int` values

Array element (an `int`) – the first of 11 in this array

Built- in Arrays

- Examples of built-in array declaration statements using constants:

```
const int NUMELS = 5; // define a constant for the number of items
int grade[NUMELS];    // declare the array

const int ARRAYSIZE = 4;
char code[ARRAYSIZE];

const int NUMELS = 6;
double prices[NUMELS];
```

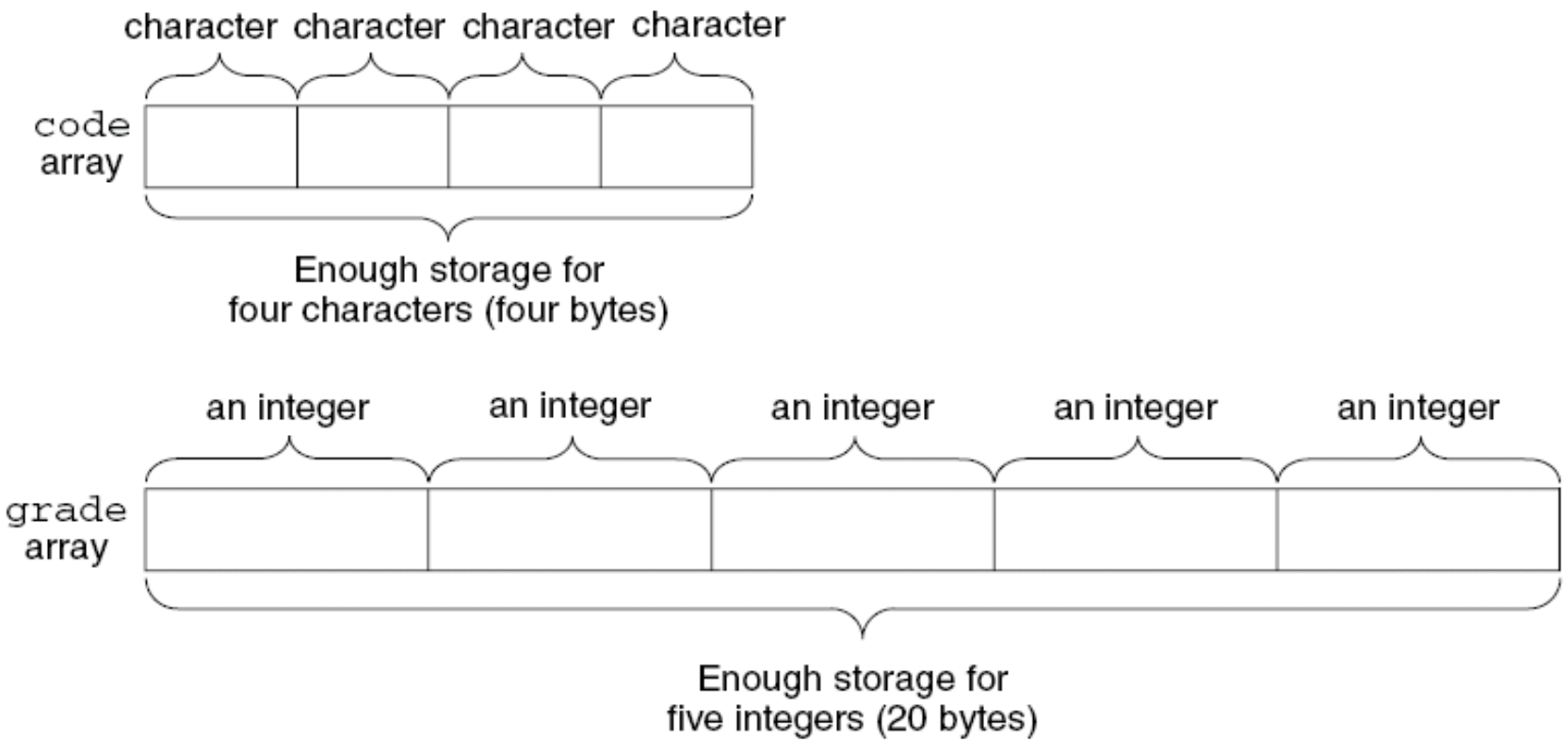
- Each array declaration statement allocates sufficient memory to hold the number of data items given in declaration

- ***Array element:*** an individual item of the array

- ***Individual array elements stored sequentially in memory***
  - A key feature of arrays that provides a simple mechanism for easily locating single elements
    - This feature is also known as contiguous storage allocation

Arrays

FIGURE 8.3 *The* `grade` *and* `code` *Arrays in Memory*



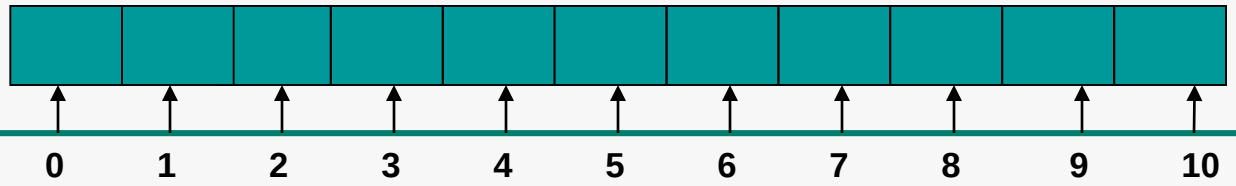
Arrays

- Looking at the Example Array

```
const int numGrades = 11;
int grades[numGrades];
```

- Key Characteristics of the array
  - Array size is fixed
    - ***Once the size of the array is declared it cannot be changed***
      - ***Exception involves pointers which we will learn later (think of this like C++ making you do the work and not having a data type do it for you like Java does – you are in control)***
  - Arrays contain homogeneous data
    - All array elements are of the same data type
  - Arrays have contiguous storage allocation
    - Individual array elements are stored sequentially
  - Arrays are Random Access
    - Array elements my be accessed individually

`grades` array: enough memory to hold 11 **`int`** values



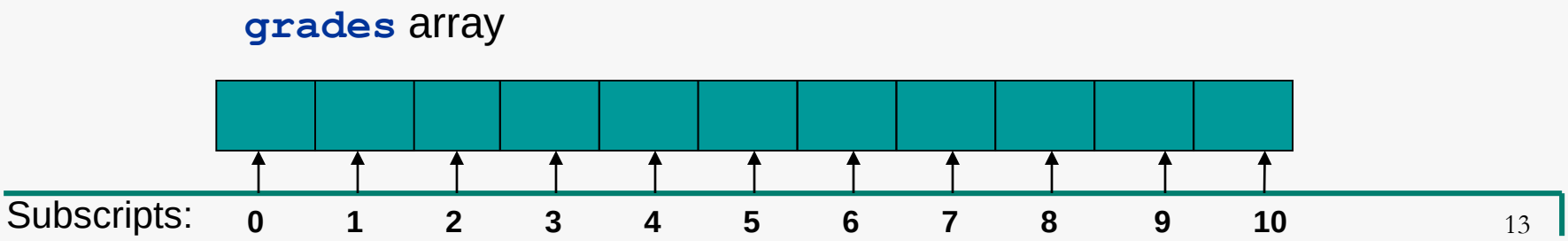
Arrays

- Arrays of any data type can be defined:

```
int days[6];                    //array of 6 ints
float temperature[100];         //array of 100 floats
char letter[26];          //array of 26 characters
double size[1200];        //array of 1200 doubles
string name[10];          //array of 10 string objects
```

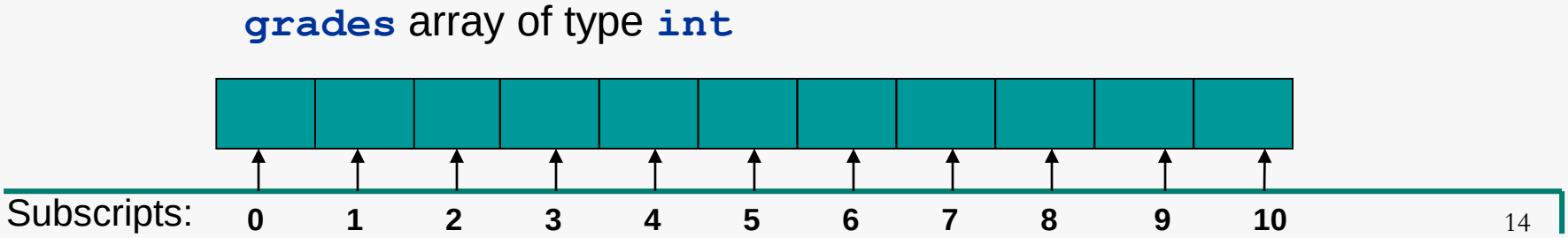
Arrays – Accessing Array Elements

- **Concept:** The individual elements of an array are assigned unique subscripts. These subscripts are used to access the individual array elements
  - Even though an array has only one name, the elements may be accessed and used as individual variables
  - Each element in an array is assigned a number known as a ***subscript (or index)***
    - A subscript is used to pinpoint a specific element in the array
    - **The first element of an array is assigned subscript 0**
      - The subscripts are assigned sequentially stepping by whole number



Arrays – Accessing Array Elements

- **Array Subscripts in C++**
  - Subscript number in in C++ always starts at zero (first array element as at subscript zero)
  - The subscript of the last element in a C++ array is one less than the total number of elements in the array
    - This array of size 11 has its last element at subscript 10



Arrays – Accessing Array Elements

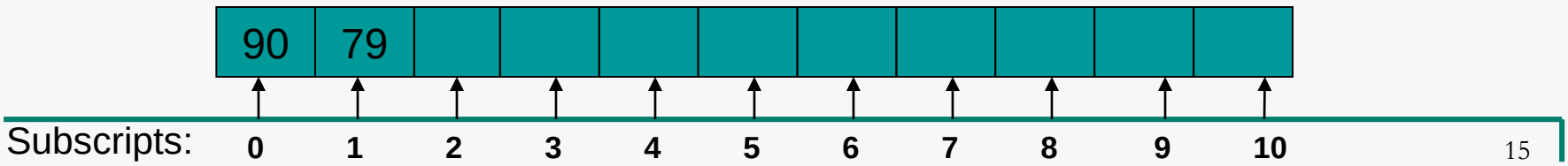
- **Each element in the array when accessed by its subscript is an individual variable of the array type**
  - So each element in the grade array is an individual **`int`** variable and can be accessed by its individual subscript

- How do we assign **`int`** values to the individual elements of the grades array?
  - The array name followed by a subscript in brackets accesses an individual element of the data type (in this case an **`int`**) of the array

```
grades[0] = 90;   //stores the number 90 in the first element of
                          //the grades array
grades[1] = 79;  //stores the number 79 in the second element of
                         //the grades array
```

  - Note that the expression grades[0] is pronounced “grades sub zero”. You would read the first assignment statement as “grades sub zero is assigned 90”

`grades` array of type **`int`**

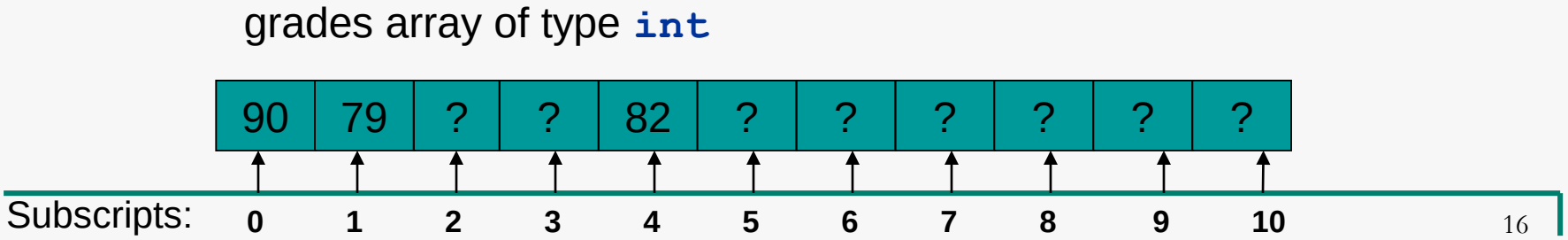


Arrays – Accessing Array Elements

- The new assignment statement assigns a value to the fifth element of the array
  - Notice the individual array elements do not need to be assigned values in order

```
grades[0] = 90;    //stores the number 90 in the first element of the grades array
grades[1] = 79;   //stores the number 79 in the second element of the grades array
grades[4] = 82;  //stores the number 82 in the fifth element of the grades array
```

- Because values have not been assigned to other elements of the array question marks have been used to indicate that the contents of those array elements are unknown. This actually depends on if an array is **global** or **local (remember we will NOT be using global anything in this class so your arrays will be local (declared inside a function)**:
  - If an array holding numeric values is defined **globally** (outside of all methods in a program), all of its elements are initialized to zero by default
  - If an array holding numeric values is defined **locally** (inside a method such as main), its elements have no default initialization value
    - So they contain whatever garbage might be in memory



Arrays – Accessing Array Elements

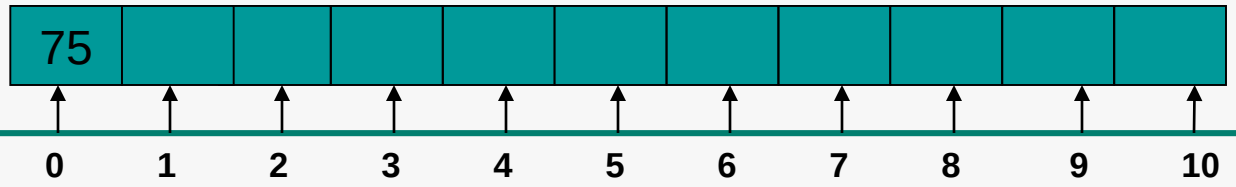
- **Subscripts vs. Size Declarators**
  - It's important to understand the difference
  - The number inside brackets in an array definition is the size declarator. It specifies how many elements the array holds.
    - This statement declares an array of 11 integer values named grades.

```
int grades[11];
```

  - The number inside brackets in an assignment statement or any statement that works with the contents of the array is a subscript.
    - This statement assigns the integer value 75 to the first element of the grades array

```
grades[0] = 75;
```

grades array of type `int`



Arrays – Accessing Array Elements

- **Arrays may NOT assign or access values for all of their elements at once**
  - They may only assign/access values one element at a time
  - Assume the following arrays have been defined:
    - Each array holds the # of patients seen by a doctor on each of 5 days

```
int doctorA[5];  //holds # of patients seen by Dr. A
int doctorB[5];  //holds # of patients seen by Dr. B
```

  - The following assignment statements are legal:

```
doctorA[0] = 31;
doctorA[1] = 40;
doctorA[2] = doctorA[0];
doctorA[0] = doctorA[1];
```

  - The following assignment statements are NOT legal:

```
doctorA = 152;      //a whole array may not be assigned a value
doctorA = doctorB;  //Array elements must be assigned one at a time
```

Arrays – Accessing Array Elements

- Inputting and Displaying Array Contents
  - The long way:

- Example (***Example1*** = arrayInputLongWay.cpp)

```
/*Asks the user to enter the grades (using numeric percentages)
 earned on a test by 6 students and stores them in an array.
It then outputs the grades (numeric percentages) entered*/

#include <iostream>
using namespace std;

int main()
{
   const int numGrades = 6;
   int grades[numGrades];  //delcares an array of 6 ints

   //read in the array contents
   cout << "Enter the grades earned on the test by six students: ";
   cin >> grades[0] >> grades[1] >> grades[2] >> grades[3]
       >> grades[4] >> grades[5];

   //Display the contents of the array
   cout << "The grades you entered are: ";
   cout << grades[0] << "," << grades[1] << "," << grades[2] << ","
        << grades[3] << "," << grades[4] << "," << grades[5] << endl;

   system("PAUSE");
   return 0;
}
```

Arrays – Accessing Array Elements

- See what prints if values aren't entered into every array element, notice that this is a local array since it's defined inside the main method
  - This can be remedied by initializing all of your array values to a default element – the default element should be something that won't be mistaken as an actual value, for this array -1 might be a good choice

- Example (***Example 2*** - localArrayUndefValues.cpp)

```
/*Asks the user to enter the grades earned on a test by first 2 students out
of 6 and stores them in a 6 element array.  This array is a local array so
when the entire array is printed out garbage is output in the array elements
no grades have been assigned to.*/

#include <iostream>
using namespace std;

int main()
{
   const int numGrades = 6;
   int grades[numGrades];  //delcares an array of 6 ints

   //read in the array contents
   cout << "Enter the grades earned on the test by the first and second student: ";
   cin >> grades[0] >> grades[1];

   //Display the contents of the array
   cout << endl;
   cout << "The grades you entered are: ";
   cout << grades[0] << "," << grades[1] << "," << grades[2] << ","
        << grades[3] << "," << grades[4] << "," << grades[5] << endl;

   system("PAUSE");
   return 0;
}
```

Arrays – Accessing Array Elements

- Even though the ***size declarator*** of an array must be a literal or constant, ***array subscript*** numbers can be stored in a variable (the variable needs to evaluate to a valid subscript)
  - This makes it possible to use a loop to cycle through an entire array performing the same operation on each element
  - This is called ***traversing*** an array
  - The grades program can be redone using a for loop

- So think about the array subscripts – They start at 0 increase by one for each consecutive array element
  - Counters in loops can start at zero and increase by 1 each loop iteration
  - We're going to take advantage of this to access array elements using a for loop

Arrays – Accessing Array Elements

- **Inputting and Displaying Array Contents**
  - **Using a loop to traverse the array and process each element:**
- ***Also let's alter this to see what happens if you try to make the size declarator a regular variable instead of a literal or named constant***

- **Example (*Example 3* - arrayInputLoop.cpp):**

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

int main()
{
   const int numGrades = 6;
   int grades[numGrades];  //delcares an array of 6 ints

   //read in the array contents
   cout << "Enter the percentages earned on the test by six students\n";
   for(int count = 0; count < 6; count++)
   {
      cout << "Please enter a grade: ";
      cin >> grades[count];
    }

   //Display the contents of the array
   cout << "The grades you entered are: ";
   for(int count = 0; count < 6; count++)
   {
      cout << grades[count];
      if(count < 5)
         cout << ",";
   }

   cout << '\n';
   system("PAUSE");
   return 0;
}
```

Arrays – Accessing Array Elements

- **Inputting and Displaying Array Contents**
  - Notice in the example program what we are doing with each loop is accessing each element of the grades array
- The for loop to get the input for the array and to output the array are both the same

```
for(int count = 0; count < 6; count++)
{
  //process each array element by accessing it with the current value of counter
  //(array[counter])
}
```

  - 6 is the size of the array and the first element of the array is at subscript zero (the value of count starts at zero)
- A better way to do this loop would have been to use the constant I have at the beginning of the program for the array size:

```
const int numGrades = 6;
for(int count = 0; count < numGrades; count++)
{
  //process each array element by accessing it with the current value of counter
  //(array[counter])
}
```

  - Why do you think this second way is better?
- What we are doing is using a loop to ***traverse*** the array and process each element:
  - The only difference between the two loops in the program is how we are processing each array element
    - One loop processes each array element by getting a value to store in it from the user and storing it there
    - The other loop processes each array element by displaying it on the screen

Arrays – Accessing Array Elements

- **Inputting and Displaying Array Contents**
  - **Using a loop to traverse the array and process each element:**

Example (*Example 4* - arrayInputLoop2.cpp):

```
/*Asks the user to enter the grades earned on a test by 6 students and stores them in an array.  It
then outputs the grades entered.  This program uses the named constant inside the for loop*/

#include <iostream>
using namespace std;

int main()
{
   const int numGrades = 6;
   int grades[numGrades];  //delcares an array of 6 ints

   //read in the array contents
   cout << "Enter the grades earned on the test by six students\n";
   for(int count = 0; count < numGrades; count++)
   {
      cout << "Please enter a grade: ";
      cin >> grades[count];
    }

   //Display the contents of the array
   cout << "The grades you entered are: ";
   for(int count = 0; count < numGrades; count++)
   {
      cout << grades[count];
      if(count < 5)
         cout << ",";
   }

   cout << '\n';
   system("PAUSE");
   return 0;
}
```

Arrays

- **Array Subscripts:** do not have to be integers
  - Any expression that evaluates to an integer may be used as a subscript
  - Subscript must be within the declared range

- Examples of valid subscripted variables (assumes `i` and `j` are `int` variables):

```
grade[i]
grade[2*i]
grade[j-i]
```

Arrays – Accessing Array Elements

- **Inputting and Displaying Array Contents**
  - **Uses symbolic constant for array size and displays the value in each array subscript**

Example (*Example 5* - arrayDisplayElements.cpp):

```
/*This program has the user enter 5 integers and then displays the
subscript of each array element and the value contained in it*/

#include <iostream>
using namespace std;

int main()
{
   const int NUM_INTS = 5;  //usually better to capitalize symbolic constants
   int grades[NUM_INTS];    //declares an array of 5 ints

   //read in the array contents
   cout << "Please enter 5 integers\n";
   for(int count = 0; count < NUM_INTS; count++)
   {
      cout << "Please enter integer #" << count + 1 << ": ";
      cin >> grades[count];
    }

   //Display the contents of the array
   cout << endl;
   cout << "The grades you entered are: \n";
   for(int count = 0; count < NUM_INTS; count++)
   {
      cout << "grades[" << count << "] = " <<grades[count] << endl;
   }

   cout << '\n';
   system("PAUSE");
   return 0;
}
```

Array Bounds Checking

- The ***bounds*** of an array are the valid subscripts of an array
  - The grades array of size 6 in the previous examples had valid subscripts from 0-5
- ***Bounds checking:*** C++ does **NOT** check if value of an array subscript used in code is within declared bounds
- If an out-of-bounds index is used in source code (say you tried to access grades[6]), C++ will NOT provide notification
  - Program will attempt to access out-of-bounds element, causing either **program error** or crash
    - These type of program errors can end up just causing unexpected results in your program that are hard to debug
  - Use symbolic constants helps avoid this problem
- A common out of bounds problem occurs when you go outside of the loop by one
- One good way to test your program is to do an echo print of the values that were input into an array like was done in the previous grades array examples

Array Bounds Checking

Example (*Example 6* - arrayBoundsError1.cpp):

```
/*This program unsafely accesses an area of memory by writing values beyond the arrays boundary.  This
could cause your program to crash*/

#include <iostream>
using namespace std;

int main()
{
   int values[3];   //this is an array with 3 elements not 4  so the elements have
                    //subscripts 0,1, and 2


   //this loop correctly inputs 100 into values[0], values[1], values[2]
   for(count = 0; count < 3; count++)
   {
     values[count] = 100;
   }

   //this loop has an error, the loop tries to display the integers in
   //values[0], values[1], values[2] and values[3]...the array does not have
   //values[3] since its size is 3...the error is the <= sign used instead of <

   cout << "The loop that prints out the array attempts to access an invalid array subscript\n";
   for(int count = 0; count <=3; count++)
   {
     cout << values[count]<< endl;
   }

   system("PAUSE");
   return 0;
}
```

Array Bounds Checking

- **Example (arrayBoundsError1.cpp) Output:**

```
The loop that prints out the array attempts to access an invalid array
   subscript

100
100
100
2293672
Press any key to continue . . .
```

- This was an array of 3 elements and the program attempted to print values for 4 elements.
  - In this example, all that happened was that garbage was printed out when the program tried to access an element not in the array
  - If we tried to input something into an invalid array subscript, the value input could potentially override something important in the area of memory outside of that reserved for the array

- Be careful with < and <= in your for loops

- **Another way to avoid errors is to use a symbolic constant for the array size**
  - (you should always do this anyway to improve program readability and to decrease the number of errors in your program should you want to change the array size later)

Array Initialization – After Array Declaration

- **Concept:** Since C++ does not initialize local array elements for you (remember the earlier example where the array elements we didn't input values in printed out garbage), it's a good idea to set each value in your array to a default value
  - That way if for some reason an array element doesn't have a value put in it you can easily tell that is the case and garbage won't exist in your program
  - Initializing each array element can be done with a for loop
  - In the grades array, since valid grades cannot be negative it might be a good idea to have this for loop execute in your program initializing all of the values to -1
    - Although my program forces the user to enter each grade, real life programs will have code that allows a user to enter one array element at a time (we'll look at this later when we start doing more complex arrays)

```
//initialize all array elements with a default value of -1
for(count = 0; count < NUM_GRADES; count++)
{
  values[count] = -1;
}
```

Array Initialization – After Array Declaration

- **Example: (*Example 7* - arrayInitializeElements.cpp)**

```
/*Asks the user to enter the grades earned on a test by the first 2
students out of 6 students and stores them in an array of 6 elements.  The
array elements have been initialized to -1 so that if the array is printed later
its easy to see the last 4 elements do not yet contain grades.
It then outputs the grades entered.*/

#include <iostream>
using namespace std;

int main()
{
   const int NUM_GRADES = 6;
   int grades[NUM_GRADES];  //declares an array of 6 ints
   int count;               //loop counter

   //initialize all of the array elements to -1
   for(count = 0; count < NUM_GRADES; count++)
   {
      grades[count] = -1;
   }
```

- **Program continued on next page**

Array Initialization – After Array Declaration

- **Example Continued: (Example 7 - arrayInitializeElements.cpp)** **Continued from previous slide**

```
  //read in the array contents for only two grade values

   cout << "Enter the grades earned on the test by the first two students\n";
   for(count = 0; count < 2; count++)
   {
      //I added a do-while loop here to not allow entry of a grade less than 0
      //or greater than 100
      //Notice since an array element is just a value of the array's data type
      //I can use it like a regular integer (or whatever data type the array is
      do
      {
        cout << "Please enter a grade: ";
        cin >> grades[count];

        if(grades[count] < 0 || grades[count] > 100)
        {
          cout << "Please enter a grade greater than 0 and less than 100\n";
        }

      }while(grades[count] < 0 || grades[count] > 100);
   }

   //Display the contents of the array
   cout << "The grades you entered are: ";
   for(count = 0; count < NUM_GRADES; count++)
   {
       //only prints array elements with a valid value in them
      if(grades[count] != -1)
      {
         cout << grades[count] << "   ";
      }
   }

   cout << '\n';
   system("PAUSE");
   return 0;
}
```

Array Initialization

- **Concept:** Arrays may be initialized when they are defined
  - Array elements can be initialized within declaration statements
  - Initializing elements must be included in braces
  - The values are stored in the array elements in the order they appear in the list
    - 19 is stored in gallons[0];
  - Example:

```
const int NUMGALS = 20;
int gallons[NUMGALS] =
   {19, 16, 14, 19, 20, 18, // initializing values
    12, 10, 22, 15, 18, 17, // may extend across
    16, 14, 23, 19, 15, 18, // multiple lines
     21, 5};
```

Partial Array Initialization

- An initialization list cannot have more values than the array has elements but it may have fewer values than there are elements
  - C++ does not require a value for every array element in the initialization list
    - However you can't "skip" elements while initializing, so if you want to initialize the element in subscript 2 for example you also have to initialize the elements in subscripts 0 and 1
- Example:

```
const int NUMVALUES = 7;
int values[NUMVALUES] = {1,2,4,8};
```

    - This definition only initializes the first four elements in the seven element array
    - ***If an array is partially initialized, the uninitialized elements will be set to zero for numeric arrays and to the null character for non-numeric arrays***
      - This is true **even if the array is a locally** defined array
        - Recall the elements of completely uninitialized **local** arrays are not assigned default values (instead contain garbage)

Array Initialization – Implicit Array Sizing

- The size of the array (size declarator) may be omitted when initializing values are included in declaration statement
- Example: the following are equivalent

```
char code[7] = {'s', 'a', 'm', 'p', 'l', 'e'};

char code[ ] = {'s', 'a', 'm', 'p', 'l', 'e'};
```

- Both declarations set aside 6 character locations for an array named **code**
  - So why is my array of size 7?
    - This is actually a C-String, a C-String is simply an array of characters, remember with C-Strings the programmer is responsible for all memory allocation
    - With a C-String the programmer is responsible for allocating memory and needs to always allocate one additional byte for the null terminator, otherwise the compiler will not know when to terminate the string
      - Show example when code array is declared as size 6 instead of size 7
- Note that you must specify an initialization list if you leave out the size declarator. Otherwise C++ doesn't know out large to make the array

Processing Array Contents

- Individual array elements are processed like any other type of variable
  - Once you access an individual element of an array using the subscript operator it's like working with any variable of that type

- Individual array elements are most often processed by traversing the array using a loop
  - The loop will use the array bounds to step through each element in the array and process it in some way
  - When we input and output values in an array in our earlier programs we were simply traversing the arrays and processing each element by either:
    - getting input and storing it into the element
    - printing out the value in each element
  - So whenever you want to traverse a one dimensional array you use the same for loop we've been discussing to visit each element of the array and do something to it (process it in some way)

Processing Array Contents

- Example: Finding the average of the values in a numeric array (Example 8 - averageGrade.cpp)

```
/*Example program to calculate the average of
some grade scores using an array*/
#include <iostream>
using namespace std;

int main()
{
   const int NUM_SCORES = 5;
   double scores[NUM_SCORES] = {90, 88, 91, 82, 95};

   double total = 0; //initialize the accumulator
   double average;   //to hold the average

   //loop to calculate(sum) the total of the grades
   for(int count = 0; count < NUM_SCORES; count++)
   {
           total += scores[count];
   }

   average = total/NUM_SCORES;

   cout << "The average of the scores is " << average << endl;

   system("PAUSE");
   return 0;
}
```

Processing Array Contents

- Example: Finding and printing all temps greater than a certain number (Example 9 - tempAve.cpp)

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

int main()
{
   const int NUM_TEMPS = 7;
   double temps[NUM_TEMPS] = {90.5, 80.7, 79.9, 75.2, 100};

   double ave;  //to store the value of an average temperature
   int numAboveAve = 0;  ///to store how many temps are above average

   do
   {
     cout << "What is the average temperature?";
     cin >> ave;

     if(ave <= 0)
       cout << "Please enter a temperature > 0";
   }while(ave <= 0);

    cout << "\nABOVE AVERAGE TEMPS: ";
   //loop print temperatures that are above average
   for(int count = 0; count < NUM_TEMPS; count++)
   {
       if(temps[count] > ave)
       {
           cout <<  temps[count] << " ";
           numAboveAve++;  //counting number of temps that were above average
       }
   }
   cout << "\nNUMBER OF TEMPS ABOVE AVERAGE: " << numAboveAve << endl << endl;

   system("PAUSE");
   return 0;
}
```

Processing Array Contents

- Other examples of programs that would traverse an array and process each element in some way
  - Find the highest value in an array
  - Find the lowest value in an array
  - Printing all elements in the array that have a specific property
  - Searching an array for a certain element (value)
  - Sum the values of a numeric array

Processing Array Contents – String Arrays

- Strings are internally stored as arrays of characters
  - They are different from other arrays in that the elements can be treated as a set of individual characters or used as a single entity
  - Normally strings are treated as single entities
    - `string name;`
    - `cout << "Enter your name: ";`
    - `cin >> name;`
    - `cout << "Hello " << name << endl;`
  - However C++ provides the ability to index arrays with subscripts
  - If we wanted to process the string **`name`** character by character we could use this type of code (this is what we were doing in the last chapter with our subscript operator for strings)
    - `cout << name[0];`
    - `cout << name[1];`

Processing String Arrays

- Example (***Example 10*** - stringArrayProc.cpp)

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

int main()
{
   char ch;
   int vowelCount = 0;
   string sentence;

   cout << "Enter a sentence: ";
   getline(cin, sentence);

   //notice that string.lengh() determines the number of chars in the string
   for(int pos = 0; pos < sentence.length(); pos ++)
   {
      //make the character in the string uppercase to process vowel switch below
      ch = toupper(sentence[pos]);

      //if the char is a vowel increment vowelcount  (you could also use a switch)
      if(ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U')
      {
            vowelCount++;
      }
   }
   cout << "There are " << vowelCount << " vowels in the sentence." << endl;

   system("PAUSE");
   return 0;
}
```

Array Initialization – Char Arrays (C Strings)

- Simplified method for initializing character arrays

```
char code[ ] = "sample"; //no braces or commas
```

- This statement uses the string "**sample**" to initialize the **code** array
  - The array is comprised of 7 characters
  - The first 6 characters are the letters:

    s, **a**, **m**, **p**, **l**, **e**
  - The last character (the **escape sequence** **\0**) is called the **Null character**

    s, **a**, **m**, **p**, **l**, **e**, **\0**

Array Elements as Function Arguments

- Array **elements** (individual array elements not entire arrays) are passed to a called function in same manner as individual regular variables

  - Example:

```
//function prototype
void showValue(int);

showValue(grades[2]); //function call

//showValue function
void showValue(int num)
{
    cout << num;
}
```

Array Elements as Function Arguments

Example: (Example 11 - arrayElementsToFunc.cpp)

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

bool isEven(int); //function prototypes
void showValue(int);

int main()
{
    const int SIZE = 5;
    int numbers[SIZE] = {25,30,16,7,-1};

    for(int counter = 0; counter < SIZE; counter++)
    {
       //array elements are passed into the functions
       if(isEven(numbers[counter]))  //same as if(isEven(numbers[counter])) == true)
       {
          showValue(numbers[counter]);
       }
    }
    cout << endl;
    system("PAUSE");
    return 0;
}

bool isEven(int value)  //function returns true if argument passed to it is even
{
     if(value%2 == 0)
     {
        return true;
     }

     return false;
}

void showValue(int value)
{
      cout << value << " ";
}
```

Array Elements as Function Arguments

- When we passed individual array elements to functions they were passed like regular variables of that array's data type
  - Remember once you access an individual element of an array, you are accessing a variable of that array's data type
    - So if you had the array:

      - `const int SIZE = 5;`
      - `int numbers[SIZE] = {25,30,16,7,-1};`

    - Whenever you access an individual element of the array using a subscript as in this example:

      - `numbers[0]`

    you are accessing a variable of the type `int` since that is the data type of an array
      - Therefore this array element can be passed into any function just like a normal `int` type variable would be.

Array Elements as Function Arguments

- **Example:** Imagine your program had a function named **`sum2Ints`** that returned the sum of two integers. How would you pass the first two elements of this array into the function?
  - This is the array declaration:
    - `const int SIZE = 5;`
    - `int numbers[SIZE] = {25,30,16,7,-1};`
  - This is the **`sum2Ints`** function

```
int sum2Ints(int value1, int value2)
{
     return value1 +  value2;
}
```

  - How would you pass the elements **`25`** and **`30`** of the **`numbers`** array to this function?
    - `sum2Ints(numbers[0], numbers[1]);`

Array Elements as Function Arguments

- **Example program (*Example 12* - sum2Ints.cpp):**

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

//function prototype
int sum2Ints(int, int);

int main()
{
    //declare the array
    const int SIZE = 5;
    int numbers[SIZE] = {25,30,16,7,-1};

    int sum2;

    //the function call - saves the return value from the function into the
    //sum2 variable
    sum2 = sum2Ints(numbers[0], numbers[1]);

    cout << "The sum is " << sum2 << endl;

    system("PAUSE");
    return 0;
}

//sum2Ints FUNCTION:  Returns the sum of the two integer arguments
int sum2Ints(int value1, int value2)
{
     return value1 +  value2;
}
```

Entire Arrays as Function Arguments

- ***In C++ by default EVERYTHING is pass by value – HOWEVER arrays are an exception to that rule***
- Passing a **complete array** into a function as an argument provides **access** to the actual array, not a copy (NOT PASS BY VALUE!)
  - Making copies of large arrays is wasteful of storage (this is why arrays are the exception to the rule)
  - **BUILT-IN Arrays in C++ are NOT objects, when you pass a built-in array to a function you pass it to a special variable that accepts the address of the array, all that variable knows is the address of the first element of the array in memory, it knows nothing about the array's size**
  - **Example:**

```
//function prototype
void showValue(int[], int);

int main()
{
   const int arraySize = 3;
   int grades[] = {1,2,3};

   showValue(grades, arraySize); //function call
}

//showValue function
void showValue(int nums[], int size)
{
      for(int index = 0; index < size; index++)
         cout << nums[index] << " ";
}
```

- Notice that along with the built-in array containing the values the function will use, the size of the array is also passed
  - This is because no size declarator is inside the brackets of **nums [ ]** in the function
  - **`nums[]` is not actually an array but a special variable that can accept the address of an array**
    - Although `nums` is not a reference variable it works like one

Entire Arrays as Function Arguments

- **Example: Passing an entire array**

```
//function prototype
void showValues(int[], int);
```

- The empty array brackets appear after the data type of the array parameter to indicate that show values accepts an address of an array, in this case an array of integers

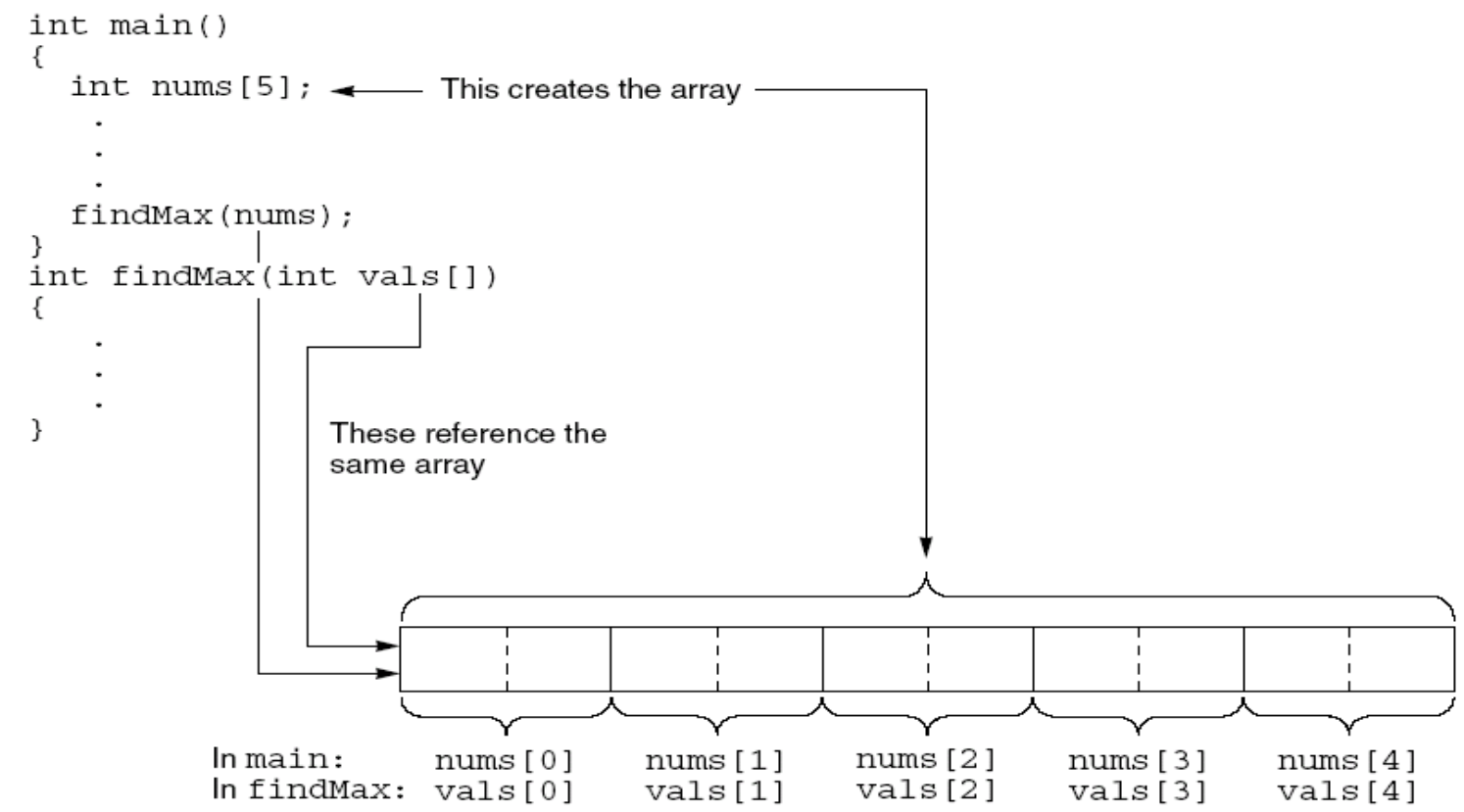
```
//function call
showValues(grades, arraySize);
```

- The first argument is the name of the array being passed to the function
  - In C++ the name of the array without brackets and a subscript is actually the beginning address of the array
- In the showValue function the beginning address of the grades array is copied into the **nums[ ]** parameter variable.
  - The nums variable is then used to reference the grades array
  - Array parameters are NOT reference variables but they work like reference variables, they give the function direct access to the array
  - **Any changes made to the the `nums[]` array parameter are made to the actual `grades` array**

```
//showValue function
void showValue(int nums[], int size)
{
       for(int index = 0; index < size; index++)
          cout << nums[index] << " ";
}
```

Entire Arrays as Arguments

FIGURE 8.7 *Only One Array Is Created*



Entire Arrays as Function Arguments

- **Example: Passing an entire array (Example 13 - passEntireGradesArray.cpp)**

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

void showValues(int[], int); //function prototype

int main()
{
   const int arraySize = 4;
   int grades[arraySize] = {91, 87, 73, 99};

   showValues(grades, arraySize);  //pass entire array to function

   system("PAUSE");
   return 0;
}

void showValues(int nums[], int size)
{
   for(int index = 0; index < size; index++)
      cout << nums[index] << " ";

    cout << endl;
}
```

Entire Arrays as Function Arguments

- **Example 14 – tripleArray.cpp:** Changes to the parameter `arr []` in the `tripleArray` function ACTUALLY change the `grades` array passed to the function

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

void tripleArray(int[], int); //function prototype

int main()
{
   const int arraySize = 4;
   int grades[arraySize] = {91, 87, 73, 99};

   tripleArray(grades, arraySize);  //pass entire array to function

   //display the array
   for(int index = 0; index < arraySize; index++)
   {
      cout << grades[index] << " ";
   }
   cout << endl;
   system("PAUSE");
   return 0;
}

void tripleArray(int arr[], int size)
{
   for(int index = 0; index < size; index++)
   {
      arr[index]  = arr[index] *3;
   }
}
```

- You can use this general idea to make all sorts of common array operations into useful functions
  - Get Highest Element, Get Lowest Element, Sum All Elements, Average All Elements

Entire Arrays as Function Arguments – Alternative Method – Example #3

- **You can pass the size of an array into the function as part of the array parameter rather than have the empty brackets and a separate parameter for the size**

- Always remember: Passing a complete array to a function provides **access** to the actual array, not a copy

  - **Example #3:**

```
//make array size a constant global variable  (EXCEPTION TO GLOBAL RULE)
const int arraySize = 4;

//function prototype
void showValues(int[arraySize]);  //no need to pass size now

showValue(grades); //function call  (no need to pass size now)

//showValue function – notice use of arraySize here
void showValue(int nums[arraySize])
{
     for(int index = 0; index < arraySize; index++)
        cout << nums[index] << " ";
}
```

- Question: Why do we have to make **`arraySize`** a global variable in this case?
  - If we didn't it would be out of scope for the function showValue

Entire Arrays as Function Arguments – Alternative Method – Example #3

Example 15 – arrayFuncAlternate.cpp

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

//make array size a constant global variable
const int arraySize = 4;

void showValues(int[arraySize]); //function prototype

int main()
{
   int grades[arraySize] = {91, 87, 73, 99};

   showValues(grades);  //pass entire array to function
   system("PAUSE");
   return 0;
}

void showValues(int nums[arraySize])
{
   for(int index = 0; index < arraySize; index++)
      cout << nums[index] << " ";

    cout << endl;
}
```

Using Parallel Arrays

- By using the same subscript you can build relationships between data stored in two or more arrays
  - Sometimes its useful to store related data in two or more arrays
  - This is especially useful when the related data is of different data types
    - Remember an array must hold data that is all the same data type
  - When data items stored in two or more arrays are related the arrays are called ***parallel arrays***

Using Parallel Arrays

- **Example:**
  - A `string` array stores an Employee's id number, a `double` array stores the hours worked by each employee and a `double` array stores each employee's hourly rate

- As you look at the example program:
  - Notice in the loops that the same subscript is used to access both arrays
    - This is because the data for an individual employee is stored in the same relative position in each array

Using Parallel Arrays – Example (Example 16 - payrollParallelArr.cpp)

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

int main()
{
  const int numEmps = 3;
  index;
  double hours[numEmps];    //parallel arrays to hold each employee's id, hours, rate
  double payRate[numEmps];
  string employeeID[numEmps];

  //get employee work data
  for(int index = 0; index < numEmps; index++)
  {
     cout << "Enter the Employee ID number:";
     getline(cin, employeeID[index]);
     cout << "Enter Hours worked by the Employee: ";
     cin >> hours[index];
     cout << "Enter Pay rate for Employee: ";
     cin >> payRate[index];
     cin.ignore();
  }
  cout << '\n';

  //display employee gross pay
  for(int index = 0; index < numEmps; index++)
  {
     cout << "The pay for Employee # " << employeeID[index] << " is: "
          << fixed << setprecision(2)
          << hours[index] * payRate[index] << endl;
  }
  system("PAUSE");
  return 0;
}
```

- **Parallel Arrays Example, the parallel arrays are employeeID, hours and payRate**

The data in the same subscripts in the three arrays relate to the same employee. For example the data in employeeID[0] hours[0] and payRate[0] all relate to the same employee.

Two Dimensional Arrays

- **Single/One Dimensional Arrays**
  - The arrays we have used are single dimensional arrays
  - A one dimensional array is used to represent items in a list or sequence of values

- **Two Dimensional Arrays**
  - Are used to represent items in a table like structure with rows and columns (provided each items in the table is of the same data type)
  - A two dimensional array is a collection of elements that are all of the same type structured in two dimensions. Each component is accessed by a pair of indices that represent the component's position in each dimension

- **Multi Dimensional Arrays**
  - Arrays can be as complex and have as many dimensions as needed

Two Dimensional Arrays

Two Dimensional Array named **`values`**

	[0]	[1]	[2]	[3]
[0]	5	3	15	0
[1]	-5	7	55	16
[2]	6	4	36	40

The Element 15 is at Row 0 Column 2

To Access the Element 15: **`values[0][2] = 15;`**

Two-Dimensional Arrays

- Two-dimensional array (table): consists of both rows and columns of elements
- **Example**: two-dimensional array of integers

```
 8      16      9     52
 3      15     27      6
14      25     2      10
```

- **The array declaration for this example**:
  - names the array
  - declares the data type of the array's elements
  - and reserves storage for the array by specifying it's size in terms of the number of rows and columns
  - HOWEVER – just like before using 3 and 4 isn't what we really should do we should use named constants for numRos and numCols – let's change it on the board

```
int values[3][4];
```

- **General syntax to declaring a two dimensional array:**

```
dataType arrayName[numRows][numColumns];
```

Two-Dimensional Arrays

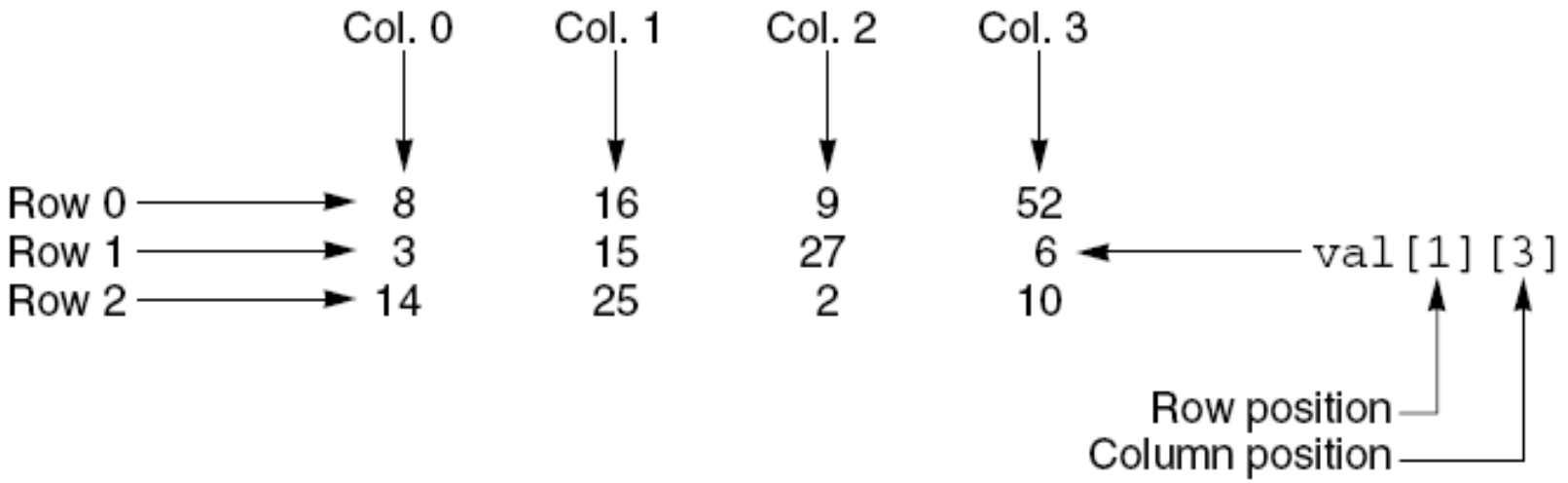
- **Accessing array elements** (Figure 8.9 – Next Slide)
  - Similar to accessing one dimensional array elements except now you need to specify both the row and column that identifies the array element you want to access – one you access an element it's like working with a regular variable of that data type
  - General syntax:
    - `arrayName[row][column]`
  - Looking at the next slide:
    `val[1][3]` uniquely identifies element in row 1, column 3
- Examples using elements of `val` array:

```
price = val[2][3];
val[0][0] = 62;
newnum = 4 * (val[1][0] - 5);
sumRow = val[0][0] + val[0][1] + val[0][2] + val[0][3];
```

  - The last statement adds the elements in row 0 and sum is stored in `sumRow`

Two-Dimensional Arrays

FIGURE 8.9 *Each Array Element Is Identified by Its Row and Column Position*



Two-Dimensional Arrays

- Array Initialization: can be done within declaration statements (as with single-dimensional arrays)
  - Also can be done with loops as we will see in a minute

- Example:

```
int val[3][4] = { {8,16,9,52},
                  {3,15,27,6},
                  {14,25,2,10} };
```

  - First set of internal braces contains values for row 0, second set for row 1, and third set for row 2
  - Commas in initialization braces are required; inner braces can be omitted
  - Sizes must be declared for each size declarator

Two-Dimensional Arrays

- ***Processing two-dimensional arrays:*** nested `for` loops typically used
  - Easy to cycle through each array element
    - A pass through outer loop corresponds to a row
    - A pass through inner loop corresponds to a column

- **Example:** Initialize a two dimensional 3x4 array as follows:

```
 2   4   6   8
10  12  14  16
18  20  22  24
```

Two-Dimensional Arrays – Example 17 – twoDimArray.cpp

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

int main()
{
   const int maxRows = 3;
   const int maxCol = 4;
   int inputNum = 0;

   int twoDim[maxRows][maxCol]; //declare array

   //input the array
   for(int row = 0; row < maxRows; row++)
   {
      for(int col = 0; col < maxCol; col++)
      {
          inputNum +=2;
          twoDim[row][col] = inputNum;
      }
   }

   //display the array
   for(int row = 0; row < maxRows; row++)
   {
      for(int col = 0; col < maxCol; col++)
      {
         cout << "[" << row << "," << col << "] = " << twoDim[row][col] << endl;
      }
   }

   system("PAUSE");
   return 0;
}
```

Two-Dimensional Arrays – Example Output

- Output:

```
[0,0] = 2
[0,1] = 4
[0,2] = 6
[0,3] = 8
[1,0] = 10
[1,1] = 12
[1,2] = 14
[1,3] = 16
[2,0] = 18
[2,1] = 20
[2,2] = 22
[2,3] = 24
Press any key to continue . .
```

- How can we rewrite the code I made to have it print in table like form? (Hint: think HW #3 – multiplication tables!)

Passing Two-Dimensional Arrays to Functions

- Two Dimensional Arrays are passed to functions almost exactly the same way one dimensional arrays are
  - Passing an array still passes an address and gives the function access to the array passed to it (so the ***ADDRESS*** is passed)

- However, prototypes for functions that pass two-dimensional arrays can omit the row size of the array but CANNOT omit the column size

  - Example:

```
Display (int nums[ ][4]);
```

  - Row size is optional but column size is required – this means we're going to want to use second method of passing an array to a function for 2D arrays
    - **This means when you pass two dimensional arrays to functions you must have at least the second size declarator passed to the function and declared in the function prototype (need this for Homework #6)**

  - In the next example I moved the display of the two dimensional array from the first example into a function to illustrate the way to pass this two dimensional array

Example (Example 18 - pass2DimArray.cpp)

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

const int maxRows = 3;  //make these global to pass 2dim array to function
const int maxCol = 4;

void displayArray(int [maxRows][maxCol]); //function prototype

int main()
{
   int inputNum = 0;
   int twoDim[maxRows][maxCol]; //declare array

   //input the array
   for(int row = 0; row < maxRows; row++)
   {
      for(int col = 0; col < maxCol; col++)
      {
         twoDim[row][col] = inputNum += 2;
      }
   }

   displayArray(twoDim);  //function call

   system("PAUSE");
   return 0;
}

/*this function accepts an entire two dimensional array as an argument and displays its contents...remember this DOES
     NOT CREATE A NEW
ARRAY or pass a copy, it provides access to the array passed to the function*/
void displayArray(int theArray[maxRows][maxCol])
{
  //display the array
   for(int row = 0; row < maxRows; row++)
   {
      for(int col = 0; col < maxCol; col++)
      {
         cout << "[" << row << "," << col << "] = " << theArray[row][col] << endl;
      }
   }
}
```

Example: (Example 19 - pass2DimArray2.cpp)

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

//make these global to pass 2dim array to function
const int maxRows = 3;
const int maxCol = 4;

void displayArray(int [maxRows][maxCol]); //function prototype
void doubleEvenElements(int [maxRows][maxCol]); //function prototype

int main()
{
   int inputNum = 0;

   int twoDim[maxRows][maxCol]; //declare array

   //input the array
   for(int row = 0; row < maxRows; row++)
   {
      for(int col = 0; col < maxCol; col++)
      {
         twoDim[row][col] = inputNum += 1;  //changed to add one rather than 2
      }
   }
    cout << "ORIGINAL ARRAY" << endl;
    displayArray(twoDim);  //function call
    cout << endl;

    doubleEvenElements(twoDim);

    cout << "ARRAY WITH EVEN ELEMENTS DOUBLED" << endl;
    displayArray(twoDim);

    system("PAUSE");
    return 0;
}
```


- **CONTINUED ON NEXT PAGE**

Example: (Example 19 - pass2DimArray2.cpp)

- **CONTINUED FROM PREVIOUS PAGE**

```
void displayArray(int theArray[maxRows][maxCol])
{
  //display the array
   for(int row = 0; row < maxRows; row++)
   {
      for(int col = 0; col < maxCol; col++)
      {
         cout << "[" << row << "," << col << "] = "
              << theArray[row][col] << endl;
      }
   }
}

//if an array element is even, this doubles it
void doubleEvenElements(int arr[maxRows][maxCol])
{
   for(int row = 0; row < maxRows; row++)
   {
      for(int col = 0; col < maxCol; col++)
      {
         if(arr[row][col] %2 == 0)
         {
            arr[row][col] *= 2;
         }
      }
   }
}
```

Larger-Dimension Arrays

- Arrays with more than two dimensions allowed in C++ but not commonly used
- Example three dimensional array declaration:

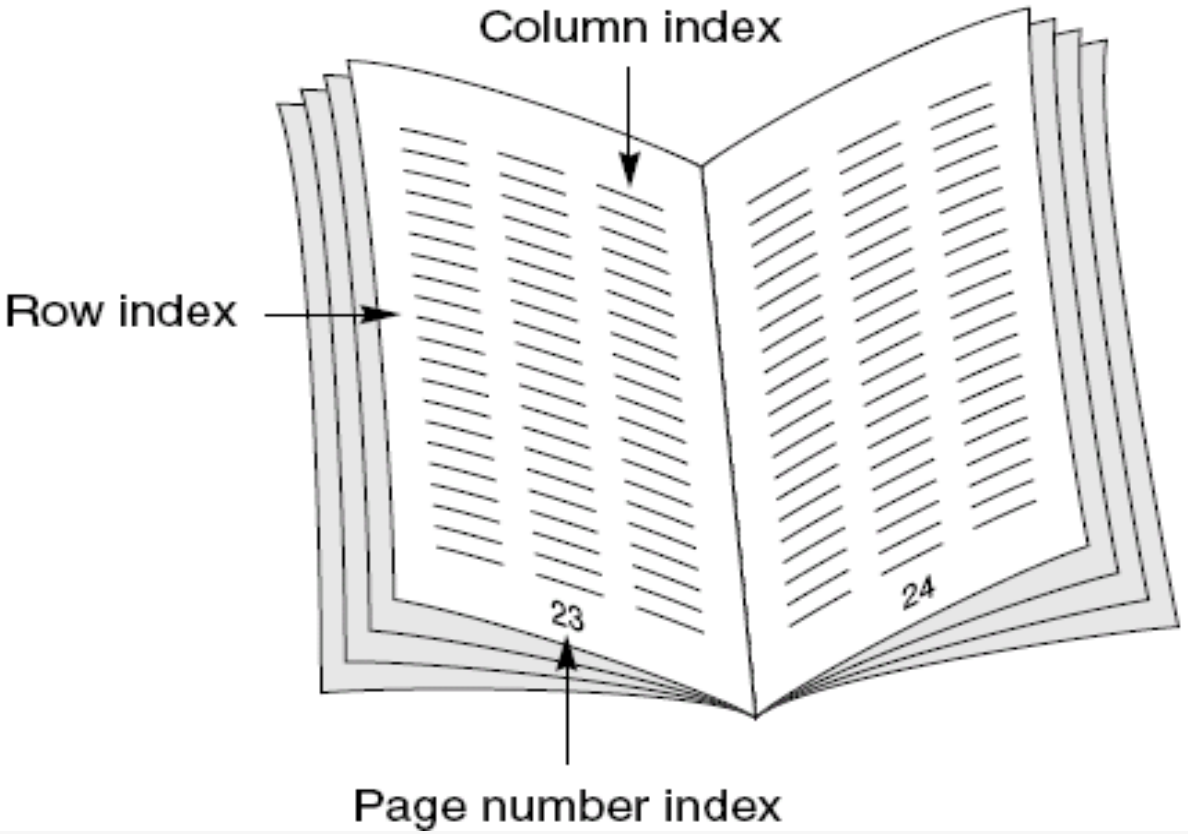
```
int response[4][10][6]
```

  - First element is `response[0][0][0]`
  - Last element is `response[3][9][5]`

- A three-dimensional array can be viewed as a book of data tables (Figure 8.12 – Next Slide)
  - First subscript (rank) is page number of table
  - Second subscript is row in table
  - Third subscript is desired column

Larger-Dimension Arrays

FIGURE 8.12 *Representation of a Three-Dimensional Array*



Common Programming Errors

- Forgetting to declare an array
  - Results in a compiler error message equivalent to “invalid indirection” each time a subscripted variable is encountered within a program
- Using a subscript that references a nonexistent array element
  - For example, declaring array to be of size 20 and using a subscript value of 25
  - Not detected by most C++ compilers and will probably cause a runtime error

Common Programming Errors

- Not using a large enough counter value in a for loop counter to cycle through all array elements

- Forgetting to initialize array elements
  - Don’t assume compiler does this

Summary

- Single-dimensional array: a data structure that stores a list of values of same data type
  - Must specify data type and array size
  - `int num[100];` creates an array of 100 integers
- Array elements are stored in contiguous locations in memory and referenced using the array name and a subscript
  - For example, `num[22]`

Summary

- Two-dimensional array is declared by listing both a row and column size with data type and name of array
- Arrays may be initialized when they are declared
  - For two-dimensional arrays you list the initial values, in a row-by-row manner, within braces and separating them with commas
- Arrays are passed to a function by passing name of array as an argument