System Design and Programming II

CSCI – 1943

David L. Sylvester, Sr., Professor

Chapter 7

Arrays

Arrays Hold Multiple Values

An array allows you to store and work with multiple values of the same data type.

An array works like a variable that can store a group of values, all of the same type. The values are stored together in consecutive memory locations.

You have previously worked with:

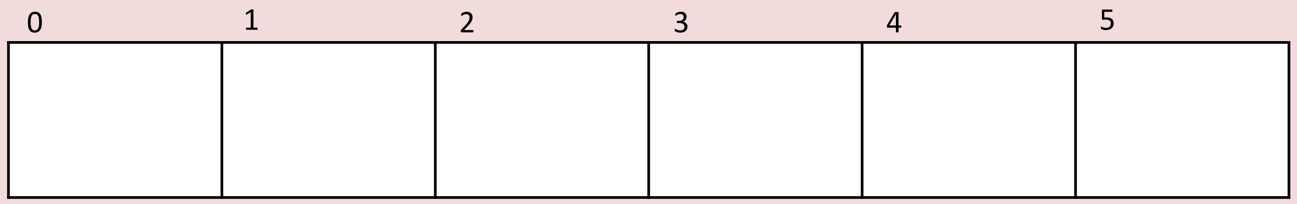
```
int days = 6;
```

days can only equal to one assigned value at a time.

In using an array:

```
int days[6];
```

Though not initialized, days can equal to six different values at one time. The number inside the brackets is the array's size declarator.



int count;	allocates enough memory for 1 int	-	12314
float price;	allocates enough memory for 1 float	-	56.981
char letter;	allocates enough memory for 1 char	-	A

Size declarator must be a constant integer expression with a value greater than zero.

int days[6];	allocates enough memory for 6 int values

An array size declarator can be either a literal (fixed value)

```
int days[6];
```

Or a named constant

```
const  int  num_days =6;
int  days[num_days];
```

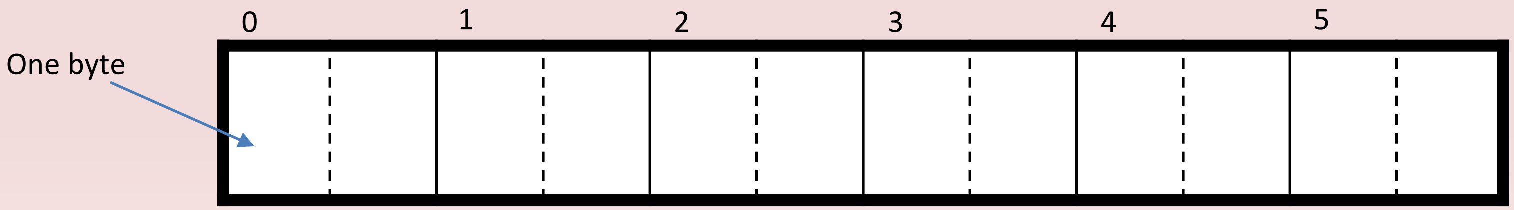
Arrays of any data type can be defined. The following are all valid array definitions.	
float temperatures[100];	Array of 100 floats
char name[41];	Array of 41 characters
long units[50];	Array of 50 long integers
double sizes[1200];	Array of 1200 doubles

Memory Requirements of Arrays

The amount of memory used by an array depends on the array's data type and the number of elements. The hours array, defined here, is an array of six shorts.

short hours[6];

On a typical PC, a short uses two bytes of memory, so the hours array would occupy 12 bytes. (Byte size of array calculated by multiplying the size of an individual element by he number of elements in the array.)



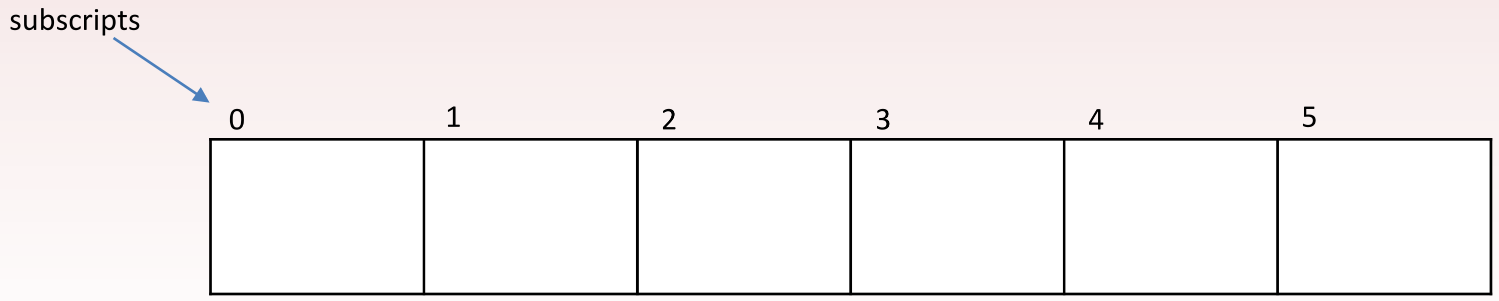
Array Definition	Number of Elements	Size of Each Element	Size of the Array
char letters[25];	25	1 byte	25 bytes
short rings[100];	100	2 bytes	200 bytes
int miles[84];	84	4 bytes	336 bytes
float temp[12];	12	4 bytes	48 bytes
double distance[1000];	1000	8 bytes	8000 bytes

Accessing Array Elements

Individual elements of an array are assigned unique subscripts. These subscripts are used to access the elements.

- An array has only one name
- Each element of the array is assigned a number called a subscript
- A subscript is used to pinpoint (locate) a specific element
- The first element is assigned the subscript 0
- Each element there after is incremented by 1

(Subscripts in C++ always starts with 0. The subscript of the last element is one less than the total number of elements in the array.)



Given the array: shorts hours[6];

You can store a value into an element of the array.

$$\overbrace{\text{hours[0]}}^{\text{Pronounced “hours sub zero”}} = 20;$$

Stores the value 20 into the first element of the hours array.

0	1	2	3	4	5
20	**?**	**?**	**?**	**?**	**?**

hours[3] = 30;

Stores the value 30 into the fourth element of the hours array.

0	1	2	3	4	5
20	**?**	**?**	**30**	**?**	**?**

Note: The number inside the brackets of an array definition is the size declarator. The number inside the brackets of an assignment statement or any statement that works with the contents of an array is a subscript.

Sample Program

Inputting and Outputting Array Content

```
// This program asks for the number of hours worked
// by six employees.  It stores the values in an array.
#include <iostream>
using namespace std;


int main()
{
        const int NUM_EMPLOYEES = 6;
        int hours[NUM_EMPLOYEES];


        //Get the hours worked by each employee.
        cout << "Enter the hours worked by "
                << NUM_EMPLOYEES << " employees: ";
        cin >> hours[0];
        cin >> hours[1];
        cin >> hours[2];
        cin >> hours[3];
        cin >> hours[4];
        cin >> hours[5];

        // Display the values in the array.
        cout << "The hours you entered are:";
        cout << " " << hours[0];
        cout << " " << hours[1];
        cout << " " << hours[2];
        cout << " " << hours[3];
        cout << " " << hours[4];
        cout << " " << hours[5] << endl;
        return 0;
}
```

Even though the size declarator of an array definition must be a constant or a literal, subscript numbers can be stored in variables. This makes it possible to use a loop to "cycle through" an entire array, performing the same operation on each element.

```
const   int   ARRAY_SIZE = 5;
int  numbers[ARRAY_SIZE];
```

Loop ends when **count** reaches 5, which is the first invalid subscript value

Variable **count** starts at 0, which is the first valid subscript value

```
for (int  count =  0; count < ARRAY_SIZE; count++)
        numbers[count] = 99;
```

Variable **count** is incremented after each iteration

Count	**ARRAY SIZE**	**numbers[count]**
0	5	99
1	5	99
2	5	99
3	5	99
4	5	99
5	5	

Sample Program Simplified By Using two for loops

```
//This program asks for the number of hours worked
// by six employees.  It stores the values in an array.
#include <iostream>
using namespace std;


int main()
{
        const int NUM_EMPLOYEES = 6;            // Number of employees
        int hours[NUM_EMPLOYEES];                       // Each employee's hours
        int count;

        //Input the hours worked.
        for (count = 0; count < NUM_EMPLOYEES; count++)
        {
                cout << "Enter the hours worked by employee "
                         << (count + 1) << ": ";
                cin >> hours[count];
        }

        // Display the contents of the array.
        cout << "The hours you entered are:" ;
        for (count = 0; count < NUM_EMPLOYEES; count ++)
                cout << " " << hours[count];
        cout << endl;
        return 0;
}
```

Loop which prompts the user for each employee hour

Displays count +1 to display number of employee.

Count is used again to step through the array, displaying each element

Integer Expressions as an Array Subscript

```
for (count = 1; count <= NUM_EMPLOYEE; count++)
{
        cout << “Enter the hours worked by employee “
             << count << “: “;
        cin >> hours[count – 1];
}
```

First time through the loop subscript equals 0.

Incorrect C++ statements **int hours[6];**

cin >> hours; (This will not work.)

Must use multiple ***cin*** *statements to read in each element.*

cout << hours; (This will not work.)

Program Exercises

Reading Data from a File into an Array (Page 491)

Writing the Contents of an Array to a File (Page 493)

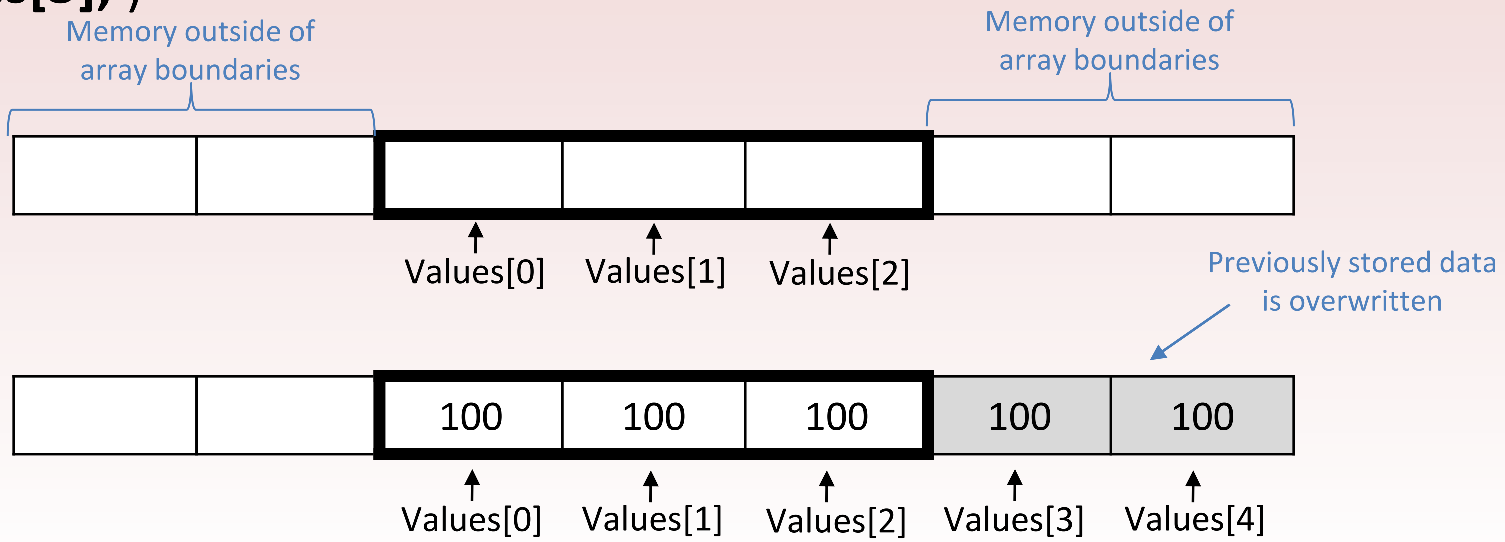
NOTE: The datafile can be created using Notepad and must be placed in the

Documents\Visual Studio 2012\Project\filename of c++ file

folder.

No Bounds Checking in C++

C++ gives you the freedom to store data past an array's boundaries. Many of the safeguards provided by other languages to prevent programs from unsafely accessing memory are absent in C++. For example, C++ does not perform array bounds checking. This means you can write programs with subscripts that go beyond the boundaries of a particular array. Be careful when attempting to compile and run programs that allow you to go beyond the boundaries of an array. This is an invalid operation, and will most likely cause the program to crash. (ex: **int values[3];**)


Memory outside of array boundaries
Memory outside of array boundaries
Values[0]
Values[1]
Values[2]
Previously stored data is overwritten
100
100
100
100
100
Values[0]
Values[1]
Values[2]
Values[3]
Values[4]

Watch for Off-by-One Errors

In working with arrays, a common type of mistake is the off-by-one error. This is an easy mistake to make because array subscripts starts at 0 rather than 1. For example, look at the following code:

```
// This code has an off-by-one error.
const  int  SIZE = 100;
Int  number[SIZE];
for (int count = 1; count <= SIZE; count++)
            numbers[count] = 0;
```

count is equal to 1.
The first element is
not accessed.

Array Initialization

Arrays may be initialized when they are defined.

Like regular variables, C++ allows you to initialize an array’s elements when you create the array.

```
const  int   MONTHS = 12;
int  days[MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

$\underbrace{\{31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31\}}_{\text{Initialization list (values separated by a comma)}}$

Or

```
const  int   MONTHS = 12;
int  days[MONTHS] = {31, 28, 31, 30, 31, 30,
                     31, 31, 30, 31, 30, 31};
```

C++ allows you to spread the initialization across multiple lines.

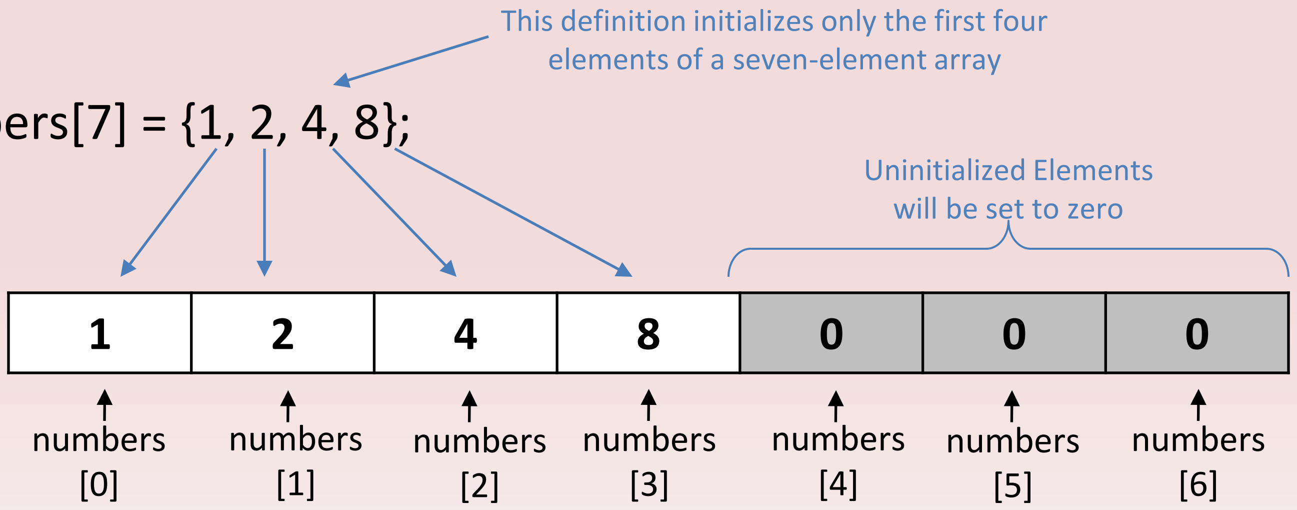
- Each value is separated by a comma

Partial Array Initialization

An array's initialization list cannot have more values than the array has elements.

When an array is being initialized, C++ does not require a value for every element. It's possible to only initialize part of an array, such as:

```
int  numbers[7] = {1, 2, 4, 8};
```


This definition initializes only the first four elements of a seven-element array
Uninitialized Elements will be set to zero
1
2
4
8
0
0
0
numbers [0]
numbers [1]
numbers [2]
numbers [3]
numbers [4]
numbers [5]
numbers [6]

Note: If you leave an element uninitialized, you must leave all the elements that follow it uninitialized as well.

Implicit Array Sizing

It is possible to define an array without specifying its size, as long as you provide an initialization list. C++ automatically makes the array large enough to hold all the initialization values.

Ex:

```
double   ratings[ ] = {1.0, 1.5, 2.0, 2.5, 3.0};
```

Because the size declarator is omitted, C++ counts the number of items in the initialization list and gives the array that many elements.

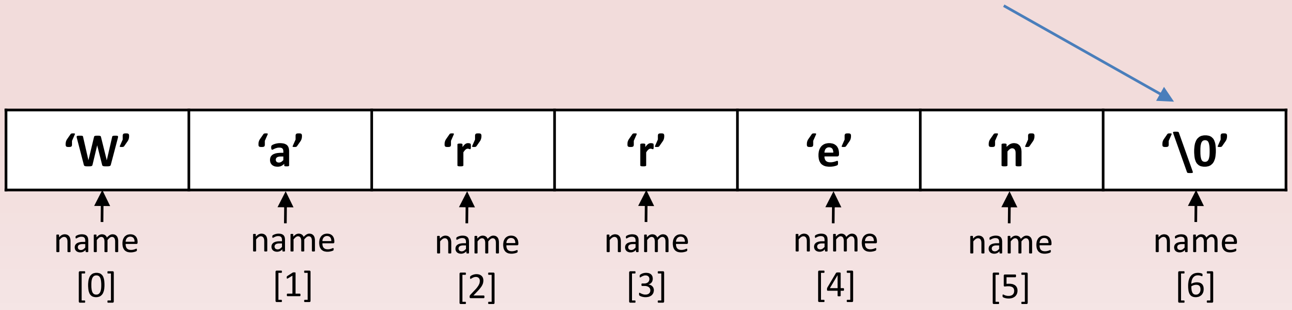
Initializing with Strings

When initializing a character array with a string, simply enclose the string in quotation marks.

Ex: `char  name[7] = “Warren”;`

– Array size for string data must accommodate for the Null Terminator.

So, even though there are six characters in the string “Warren,” the array must have enough space to accommodate the null terminator at the end of the string.



Note: ‘\0’ represents the null terminator. It is an escape sequence that is stored in memory as a single character. The null terminator is not automatically included when an array is initialized with individual characters. It must be included in the initialization list.

Ex: `char name[7] = {‘W’,  ‘a’,  ‘r’,  ‘r’,  ‘e’,  ‘n’,  ‘\0’};`

Array Initialization

String Literal and Individual Character

```
//This program display the contents of two char arrays.
#include <iostream>
using namespace std;

int main()
{
        char name1[ ] = "Holly";
        char name2[ ] = {'W', 'a', 'r', 'r', 'e', 'n', '\0'};

        cout << name1 << endl;
        cout << name2 << endl;
        return 0;
}
```

Size declarator not specified; compiler will size the array to hold initialized value

name1 has six elements (**five characters and a null terminator**)

name2 has seven elements

Processing Array Content

Individual array elements are processed like any other type of variable.

Ex: pay = $\underbrace{\text{hours[3]}}$ * rate;

fourth element of hours array is multiplied
by **rate** and results stored into **pay**

Examples of pre-increment and post-increment operations on array elements.

```
int  score[5] = {7, 8, 9, 10, 11};
++score[2];          //  Pre-increment operation on the value in score[2]
```

(Will result in adding 1 to 9 giving 10 before computing the expression.)

```
score[4]++;          // Post-increment operation on the value in score[4]
```

(Will result in adding 1 to 11 giving 12 after computing the expression.)

Be careful when using increment and decrement operators.

amount[count--]; $\longleftarrow$ Post-decrement that decrements the **subscript** of the array amount.

amount[count]--; $\longleftarrow$ Post-decrement that decrements the **value of the element** in the array amount.

Program: Calculate gross pay for five employees

```
// This program stores, in an array, the hours worked by
// employees who all make the same hourly wage.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
        const int NUM_EMPLOYEES = 5;
        int hours[NUM_EMPLOYEES];
        double payrate;

        // Input the hours worked.
        cout << "Enter the hours worked by ";
        cout << NUM_EMPLOYEES << " employees who all\n";
        cout << "earn the same hourly rate.\n";
        for (int index = 0; index < NUM_EMPLOYEES; index++)
        {
                cout << "Employee #" << (index + 1) << ": ";
                cin >> hours[index];
        }
 // Input the hourly rate for all employees.
        cout << "Enter the hourly pay rate for all the employees: ";
        cin >> payrate;

        // Display each employee's gross pay.
        cout << "Here is the gross pay for each employee:\n";
        cout << fixed << showpoint << setprecision(2);
        for (int index = 0; index < NUM_EMPLOYEES; index++)
        {
                double grossPay = hours[index] * payrate;
                cout << "Employee #" << (index + 1);
                cout << ": $" << grossPay << endl;
        }
return 0;
}
```

Array element can also be used in relational expressions.

```
        Ex:  if (cost[20] < cost[0])

                 while (value[place] !=  0)
```

Thou Shall not Assign

```
const  int  SIZE = 4;
int  oldValues[SIZE] = {19, 100, 200, 300};
int  newValues[SIZE];

newValues = oldValues;
```

This is wrong and will never work because you cannot change the starting memory address of an array.

The only way to assign one array to another is to assign individual elements in the array.

```
for (int count = 0; count < SIZE; count++)
        newValues[count] = oldValues[count];
```

Printing the Contents of an Array

Given the following:

```
const  int  SIZE = 5;
int  array[SIZE] = {10, 20, 30, 40, 50};

cout << array
```

This is wrong and will never work because the array name is seen as the beginning memory address; not the address of its contents.

The only way to output the array contents is to use a loop.

```
for (int count = 0; count < SIZE; count++)
        cout << array[count] << endl;
```

There is one exception to this rule when it comes to character arrays.

```
char  name[ ] = “Ruth”;
cout  <<  name << endl;
```

This is because the stream insertion operator is designed to behave differently when it receives the address of a char array; when received, it assumes a C-string is stored at that address, and sends the C-string to cout.

Finding the Highest and Lowest Values in a Numeric Array

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

// Declaring variable that will be used in the program
int highest, lowest;

int main()
{
// Defining and initializing array
const int array_size = 5;
int numbers[array_size] = {5211, 200, 33, 4444, 12};

// Loop that finds the highest value
highest = numbers[0];
for (int count = 0; count < array_size; count++)
{
if (numbers[count] > highest)
highest = numbers[count];
}

// Loop that finds the lowest value
lowest = numbers[0];
for (int count = 0; count < array_size; count++)
{
if (numbers[count] < lowest)
lowest = numbers[count];
}

// Prints out the highest and lowest value of the
array
cout << endl << "Highest is: " << highest << endl;
cout << endl << " Lowest is: " << lowest << endl;
return 0;
}
```

Partially Filled Arrays

Sometimes you need to store a series of items in an array, but you do not know the number of items there are. As a result, you do not know the exact number of elements needed for the array.

One solutions is to make the array large enough to hold the largest possible number of items. This can leads to another problem; a partially filled array. You would have to make sure that you only process the elements that contains valid data items.

To fix this problem, an integer variable is normally use to store the number of items in the array.

Ex:

```
const  int  SIZE= 100;
int  array[SIZE];
int count = 0;


int  number;
cout << “Enter a number or -1 to quit: “;
cin >> number;
while (number != -1   &&   count < SIZE)
{
            count++;
            array[count – 1] = number;
            cout << “Enter a number or -1 to quit: “;
            cin >> number;
}
```

The loop ends when the user enters -1 or count exceeds 99

count is integer variable that is incremented each time a value other than -1 is entered

Program: Read Input File into Array

```
//  This program reads data from a file into an array.
#include <iostream>
#include <fstream>
using namespace std;


int main()
{
        const int ARRAY_SIZE = 100;             // Array size
        int numbers[ARRAY_SIZE];                // Array with 100 elements
        int count = 0;                          // Loop counter variable
        ifstream inputFile;                     // Input file stream object
        inputFile.open("numbers.txt");          // Open the file.


         // Read the numbers from the file into the array.
        // After this loop executes, the count variable will hold
        // the number of values that were stored in the array.
        while (count < ARRAY_SIZE && inputFile >> numbers[count])
                count++;


        // Close the file.
        inputFile.close();


        // Display the number read.
        cout << "The numbers are: " << endl;
        for (int index = 0; index < count; index++)
                cout << numbers[index] << endl;
        cout << endl;
         return 0;
}
```

The loop ends when count becomes greater than ARRAY_SIZE or EOF is encountered (Both conditions must be true for the loop to process.

Based on the previous loop, **count** become the max size of the numbers array.

Comparing Arrays

- Cannot assign one array to another
- Cannot use the = = operator to compare arrays

Ex:

```
int  firstArray[ ] = {5, 10, 15, 20, 25};
int  secondArray[ ] = {5, 10, 15, 20, 25};
if (firstArray  = =  secondArray)
        cout << "The arrays are the same.\n";
else
        cout << "The arrays are not the same\n";
```

This comparison is invalid. It attempts to compare the memory address or each array

To compare the contents of two arrays, you must compare the elements of the two arrays.

Program: Comparing Two Arrays

```
//  This program compares two arrays.
#include <iostream>
using namespace std;


int main()
{
        const int SIZE = 5;// Array size
        int firstArray[SIZE] = {5, 10, 15, 20, 25};// Array with 5 elements
        int secondArray[SIZE] = {45, 10, 15, 20, 25};// Array with 5 elements
        bool arraysEqual = true;// Flag variable
        int count = 0;// Loop counter variable


        // Determine whether the elements contain the same data.
       while (arraysEqual && count < SIZE)
        {
                if (firstArray[count] != secondArray[count])
                        arraysEqual = false;
                count++;
        }


        if (arraysEqual)
                cout << "The arrays are equal.\n";
        else
                cout << "The arrays are not equal.\n";


        return 0;
}
```

Boolean operator set to true; is set to false when elements are not equal

Program: Using Parallel Arrays

By using the same subscript, you can build relationships between data stored in two or more arrays.

Sometimes it is useful to store related data in two or more arrays. It's especially useful when the related data is of unlike types. For example, you may want two arrays to store the hours worked by each employees; as **int**, and another to store each employee's hourly pay rate; as **double**.

Ex:

```
for (int index = 0; index < NUM_EMPLOYEES; index++)
{
        cout << "Hours worked by employee #" << index + 1 << ": ";
        cin >> hours[index];
        cout << "Hourly pay rate for employee #" << index + 1 << ": ";
        cin >> payRate[index];
}
```

Both arrays (**hours** and **payRate**) use the same index value to store values

Program: Comparing Two Arrays

```
// This program uses two parallel arrays: one for hours
// worked and one for pay rate.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
	const int NUM_EMPLOYEES = 5;// Number of
	employees
	int hours[NUM_EMPLOYEES];// Holds hours worked
	double payRate[NUM_EMPLOYEES];// Holds pay rate
	bool arraysEqual = true;// Flag variable

	// Input the hours worked and the hourly pay rate.
	cout << "Enter the hours worked by " <<
	 NUM_EMPLOYEES
	 << " employees and their\n"
	 << "hourly pay rate.\n";
```

Both arrays (**hours** and **payRate**) use the same index value to store values

```
	for (int index = 0; index < NUM_EMPLOYEES; index++)
	{
		cout << "Hours worked by employee #" << (index +
			1) << ": ";
		cin >> hours[index];
		cout << "Hourly pay rate for employee #" << (index
			+ 1) << ": ";
		cin >> payRate[index];
	}


	// Display each employee's gross pay.
	cout << "Here is the gross pay for each employee:\n";
	cout << fixed << showpoint << setprecision(2);
	for (int index = 0; index < NUM_EMPLOYEES; index++)
	{
		double grossPay = hours[index] * payRate[index];
		cout << "Employee #" << (index + 1);
		cout << ": $" << grossPay << endl;
	}
	return 0;
}
```

Arrays as Function Arguments

```
// This program demonstrated that an array element is passed
// to a function like any other variable
#include <iostream>
using namespace std;


void showValue(int);// Function prototype


int main()
{
        const int SIZE = 8;
        int numbers[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};


        for (int index = 0; index < SIZE; index++)
        showValue(numbers[index]);
        return 0;
}


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

Each time **showValue** is called, a copy of an array element is passed into the parameter variable num. (Array element is passed)

Definition of function **showValue**. This function accepts an integer argument. The value of the argument is displayed.

Passing Entire Array in Function Argument

```
// This program demonstrates an array being passed to a function.
#include <iostream>
using namespace std;
int total;
void showValues(int [  ], int);     // Function prototype

int main()
{
	const int ARRAY_SIZE = 8;
	int numbers[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};

	showValues(numbers, ARRAY_SIZE);
	return 0;
}

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

Notice that the brackets are empty. This is because the **showValues** function will accept the address of an array of integer

When showValues is called, the address of the array (**numbers**) and the array size (**ARRAY_SIZE**) is passed to the showValues function.

The beginning address of the **numbers** array is copied into the **nums** parameter

Is actually printing **numbers**[0-7]

Passing Two Arrays in Function Argument

```
// This program demonstrates the showValues function being
// used to display the contents of two arrays.
#include <iostream>
using namespace std;

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

int main()
{
	const int SIZE1 = 8;// Size of set1 array
	const int SIZE2 = 5;// Size of set2 array
	int set1[SIZE1] = {5, 10, 15, 20, 25, 30, 35, 40};
	int set2[SIZE2] = {2, 4, 6, 8, 10};

	// Pass set1 to showValues.
	showValues(set1, SIZE1);

	// Pass set2 to showValues.
	showValues(set2, SIZE2);
	return 0;
}

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

When showValues is called, the address of the array (**set1**) and the array size (**SIZE1**) is passed to the showValues function.

When showValues is called, the address of the array (**set2**) and the array size (**SIZE2**) is passed to the showValues function.

Array parameters work very much like reference variables. They give the function direct access to the original array. Any changes made with the array parameter are actually made on the original array used as the argument.

Two Dimensional Arrays

A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data.

Sometimes it becomes necessary to work with multiple sets of data. For example, in a grade-averaging program a teacher might record all of one student's test scores in an array of **double**s. If a teacher has 30 students, that means she'll need 30 arrays of doubles to record the score for the entire class. Instead of defining 30 individual arrays, however, it would be better to define a two dimensional array.

- One dimensional arrays can only hold one set of data.
- Two-dimensional arrays (2D arrays) hold multiple sets of data
  - Having rows and columns of elements

The array has three rows (0 – 2) and four columns (0 – 3). There is a total of 12 elements in the array.

Ex:

	Column 0	Column 1	Column 2	Column 3
Row 0	scores [0] [0]	scores [0] [1]	scores [0] [2]	scores [0] [3]
Row 1	scores [1] [0]	scores [1] [1]	scores [1] [2]	scores [1] [3]
Row 2	scores [2] [0]	scores [2] [1]	scores [2] [2]	scores [2] [3]

Defining a two-dimensional array

- Two size declarators are required
  - First one for the number of rows
  - Second for the number of columns

Ex:

```
double scores[3] [4];
```

Row Column

When processing the data in a two-dimensional array, each element has two subscripts: one for its row and another for its column.

Elements of row 0	Elements of row 1	Elements or row 2
scores[0][0]	scores[1][0]	scores[2][0]
scores[0][1]	scores[1][1]	scores[2][1]
scores[0][2]	scores[1][2]	scores[2][2]
scores[0][3]	scores[1][3]	scores[2][3]

The subscripted references are used in a program just like the references to elements in a single-dimension array, except now you use two subscripts.

Ex:

```
scores[2][1] = 92.25;
```

Stores the value 92.25 into the element at row 2, column 1 of the scores array.

```
cout << scores[0][2];
```

Displays the element at row 0, column 2.

Program: Two Dimensional Array

```
// This program demonstrates a two-dimensional
array.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
const int NUM_DIVS = 3;// Number of divisions
const int NUM_QTRS = 4;// Number of quarters
double sales[NUM_DIVS][NUM_QTRS];// Array
with 3 rows and 4 columns.
double totalSales = 0;// To Hold the total sales.
int div, qtr;// Loop counters.

cout << "This program will calculate the total sales
of \n";
cout << "all the company's division.\n";
cout << "Enter the following sales
information:\n\n";
```

```
// Nested loops to fill the array with quarterly
// sales figures for each division.
for (div = 0; div < NUM_DIVS; div++)
{
for (qtr = 0; qtr < NUM_QTRS; qtr++)
{
cout << "Division " << (div + 1);
cout << ", Quarter " << (qtr + 1) << ": $";
cin >> sales[div][qtr];
}
cout << endl; // Print blank line.
}

// Nested loops used to add all the elements.
for (div = 0; div < NUM_DIVS; div++)
{
for (qtr = 0; qtr < NUM_QTRS; qtr++)
totalSales += sales[div][qtr];
}

cout << fixed << showpoint << setprecision(2);
cout << "The total sales for the company are: $";
cout << totalSales << endl;
return 0;
}
```

Initializing Two-Dimensional Arrays

When initializing a two-dimensional array, it helps to enclose each row's initialization list in a set of brackets.

Ex:
```
int  hours[3][2] = { {8,5}, {7,9}, {6,3} };
```

Or

```
int hours[3][2] = {{8,5},
                   {7,9},
                   {6,3}};
```

In either case, the values are assigned to hours in the following manner:

hours[0][0]	is set to	8
hours[0][1]	is set to	5
hours[1][0]	is set to	7
hours[1][1]	is set to	9
hours[2][0]	is set to	6
hours[2][1]	is set to	3

	Column 0	**Column 1**
Row 0	8	5
Row 1	7	9
Row 2	6	3

The extra brackets that enclose each row’s initialization list are optional.

Ex: int hours[3][2] = {{8,5}, {7,9}, {6,3}};

or

int hours[3][2] = {8, 5, 7, 9, 6, 3};

Both statements perform the same initialization.

Because the extra braces usually separate each row, it’s a good idea to use them. In addition, braces give you the ability to leave out initializers within a row without omitting the initializers for the rows that follow it.

Ex: int table[3][2] = {{1}, {3,4}, {5}};

In the above initialization, table[0][0] is initialized to **1**, table [1][0] is initialized to **3**, table [1][1] is initialized to **4**, and table [2][0] is initialized to **5**. table[0][1] and table[2][1] are not initialized.

Because table[0][1] and table [2][1] were not initialized, they are automatically set to zero.

Passing Two-Dimensional Arrays to Functions

```
// This program demonstrates accepting a 2D array
argument.
#include <iostream>
#include <iomanip>
using namespace std;


// Global constants
const int COLS = 4;// Number of columns in each array
const int TBL1_ROWS = 3;// Number of rows in Table1
const int TBL2_ROWS = 4;// Number of rows in Table2


void showArray(int [][COLS], int);//Function prototype


int main()
{
int table1[TBL1_ROWS][COLS] = {{1, 2, 3, 4},
  {5, 6, 7, 8},
  {9, 10, 11, 12}};
int table2[TBL2_ROWS][COLS] = {{10, 20, 30, 40},
  {50, 60, 70, 80},
  {90, 100, 110, 120},
  {130, 140, 150, 160}};
```

COLS is a global name constant which is set to 4.

```
cout << "The contents of table1 are:\n";
showArray(table1, TBL1_ROWS);
cout << "The contents of table2 are:\n";
showArray(table2, TBL2_ROWS);
return 0;
}


void showArray(int array1[][COLS], int rows)
{
for (int x = 0; x < rows; x++)
{
for (int y = 0; y < COLS; y++)
{
cout << setw(4) << array1[x][y] << " ";
}
cout << endl;
}
}
```

The functions accept any two dimensional integer array, as long as it consists of four columns.

Array Programs

- Summing the Rows of a Two Dimensional Array
- Summing the Column of a Two Dimensional Array
  - Calculating average of each column

Arrays of Strings

A two-dimensional array of characters can be uses as an array of strings.

Because strings are stored in single-dimensional character arrays, an array of strings would be a two-dimensional character array.

```
Ex:        char scientists[4][9] = {“Galileo”,
                                    “Kepler”,
                                     “Newton”,
                                     “Einstein”};
```

G	a	l	i	l	e	o	\0	
K	e	p	l	e	r	\0		
N	e	w	t	o	n	\0		
E	i	n	s	t	e	i	n	\0

Longest string in the array has nine characters including null terminator. The row with strings of less than nine characters will have unused elements.

Just as the name of an array represents the array’s address, a two-dimensional array with only the row subscript represents the address of the row. (*scientist[0] represents the address of row 0*) Galileo

Arrays of Strings

The following statement will display “Einstein”.

```
cout << scientists[3];
```

The following loop will display all the names in the array.

```
for (int count = 0; count < 4; count ++)
        cout << scientist[count] << endl;
```

```
// This program displays the number of days in each month.
#include <iostream>
using namespace std;

int main()
{
const int NUM_MONTHS = 12;// The number of months
const int STRING_SIZE = 10;// Maximum size of each string

// Array with the names of the months
char months[NUM_MONTHS][STRING_SIZE]=
{ "January", "February", "March",
  "April", "May", "June",
  "July", "August", "September",
  "October", "November", "December" };
```

```
// Array with the number of days in each month
int days[NUM_MONTHS] = {31, 28, 31, 30,
31, 30, 31, 31,
30, 31, 30, 31};

// Displays the months and their numbers of days.
for (int count = 0; count < NUM_MONTHS; count++)
{
cout << months[count] << " has ";
cout << days[count] << " days.\n";
}
}
```

Arrays with Three or More Dimensions

C++ does not limit the number of dimensions that an array may have. It is possible to create arrays with multiple dimensions, to model data that occur in multiple sets.

Example of three dimensional array definition:

double seats[3][5][8];

This array can be thought of as three sets of five rows, with each row containing eight elements. Could be used to store the prices of seats in an auditorium, where there are eight seats in a row, five rows in a section and a total of three sections.

Introduction to STL Vector

The Standard Template Library offers a vector data type, which in many ways, is superior to standard arrays.

The Standard Template Library(STL) is a collection of data types and algorithms that you might use in your programs. These data types and algorithms are programmer-defined. They are not part of the C++ language, but were created in addition to the built-in data types.

The data types that are defined in the STL are commonly called containers. They are call containers because they store and organize data. There are two types of containers in the STL: sequence containers and associative containers. A sequence container organizes data in a sequential fashion, similar to an array. Associative containers organize data with keys, which allow rapid, random access to elements stored in the container.

A vector is like an array in the following ways:

- A vector holds a sequence of values, or elements.
- A vector stores its elements in contiguous memory locations.
- You can use the array subscript operator [ ] to read the individual elements in the vector.

Advantages of using vectors over arrays:

- You do not have to declare the number of elements that a vector will have
- If you add a value to a vector that is already full, the vector will automatically increase its size to accommodate the new value.
- Vectors can report the number of elements it contains.

Defining a vector

To use vectors in your program, you must include the vector header file.

Ex: `#include <vector>`

Defining a vector:

Ex: `vector<int>  numbers;`

This statement declares a vector named numbers of int datatype with no size.

Defining a vector with a size:

Ex: `vector<int> numbers(10);`

Notice that you have the same statement as the previous one, with the addition of the size in parenthesis.

Unlike arrays, you can initialize a vector with the values in another vector.

Ex: `vector<int>  set2(set1);`

After the execution of this statement, set2 will be a copy of set1.

Definition Format	**Description**
vector<float> amounts;	Defines amounts as an empty vector of floats
vector<int> scores(15);	Defines scores as a vector of 15 ints
vector<char> letters(25, ’A’);	Defines letters as a vector of 25 characters. Each element is initialized with ‘A’.
vector<double> values2(values1)	Defines values2 as a vector of doubles. All the elements of values1, which is also a vector of doubles, are copied to value2.

To store a value in an element that already exist in a vector, you may use the array subscript operator [   ].

Storing and Retrieving Values in a vector

```
#include <iomanip>// Needed to set decimal place in output
#include <vector>// Needed to define vectors
using namespace std;

int main()
{
   const int NUM_EMPLOYEES = 5;// Number of employees
   vector<int> hours(NUM_EMPLOYEES);// A vector of integers
   vector<double> payRate(NUM_EMPLOYEES);
   int index;// Loop counter

   // Input the data.
   cout << "Enter the hours worked by " << NUM_EMPLOYEES;
   cout << " employees and their hourly rates.\n";
   for (index = 0; index < NUM_EMPLOYEES; index++)
   {
       cout << "Hours worked by employee #" << (index + 1);
       cout << ": ";
       cin >> hours[index];
       cout << "Hourly Pay rate for employee #";
       cout << (index + 1) << ": ";
       cin >> payRate[index];
   }
```

vectors with starting size of 5

Subscript operator [ ] can be used because vector elements already exists

Loop that stores values into each element of the vectors

```
   // Display each employee's gross pay.
   cout << "\nHere is the gross pay for each employee:\n";
   cout << fixed << showpoint << setprecision(2);
   for (index = 0; index < NUM_EMPLOYEES; index++)
      {
           double grossPay = hours[index] * payRate[index];
           cout << "Employee #" << (index + 1);
           cout << ": $" << grossPay << endl;
      }
      return 0;
}
```

Using push_back Member Function

You cannot use the [ ] operator to access a vector element that does not exist. To store a value in a vector that does not have a starting size, or that is already full, use the push_back member function. The push_back member function accepts a value as an argument, and stores that value after the last element in the vector. (i.e. Pushes the value onto the back of the vector.)

Ex:
```
numbers.push_back(25);
```

Assuming numbers is a vector of ints, this statement stores 25 as the last element. This means, if numbers is full, the statement creates a new last element, and stores 25 in it. If there are no element in numbers, this statement creates an element and stores 25 in it.

```
// This program stores, in two vectors, the
// worked by 5 employees, and their hourly pay rate.
#include <iostream>
#include <iomanip>// Needed to set decimal place
#include <vector>// Needed to define vectors
using namespace std;

int main()
{
    vector<int> hours;// A vector of integers
    vector<double> payRate;// A vector of doubles
    int numEmployees;// The number of employees
    int index;// Loop counter


    // Input number of employees.
    cout << "How many employees do you have? ";
   cin >> numEmployees;

    // Input the payroll data
   cout << "Enter the hours worked by " << numEmployees;
   cout << " employees and their hourly rate.\n";
```

```
     for (index = 0; index < numEmployees; index++)
    {
        int tempHours;
        double tempRate;
        cout << "Hours worked by employee #" << (index + 1);
        cout << ": ";
        cin >> tempHours;
        cout << "Hourly Pay rate for employee #";
        cout << (index + 1) << ": ";
        cin >> tempRate;
        hours.push_back(tempHours);
        payRate.push_back(tempRate);
     }


    // Display each employee's gross pay.
    cout << "\nHere is the gross pay for each employee:\n";
    cout << fixed << showpoint << setprecision(2);
    for (index = 0; index < numEmployees; index++)
   {
        double grossPay = hours[index] * payRate[index];
        cout << "Employee #" << (index + 1);
        cout << ": $" << grossPay << endl;
    }
    return 0;
}
```

Determining the Size of a vector

Unlike arrays, vectors can report the number of elements they contain. This is accomplished with the size member function.

Ex:

```
numValue  = set.size();
```

numValue is of int and set is a vector. numValue will contain the number of element in the set.

The size function is useful when writing functions that accept vectors as arguments.

Ex:

```
void showValue(vector<int> vect)
{
        for (int count = 0; count < vect.size(); count++)
                count << vect[count] << endl;
}
```

Because the vector can report its size, this function does not need a second argument indicating the number of elements in the vector.