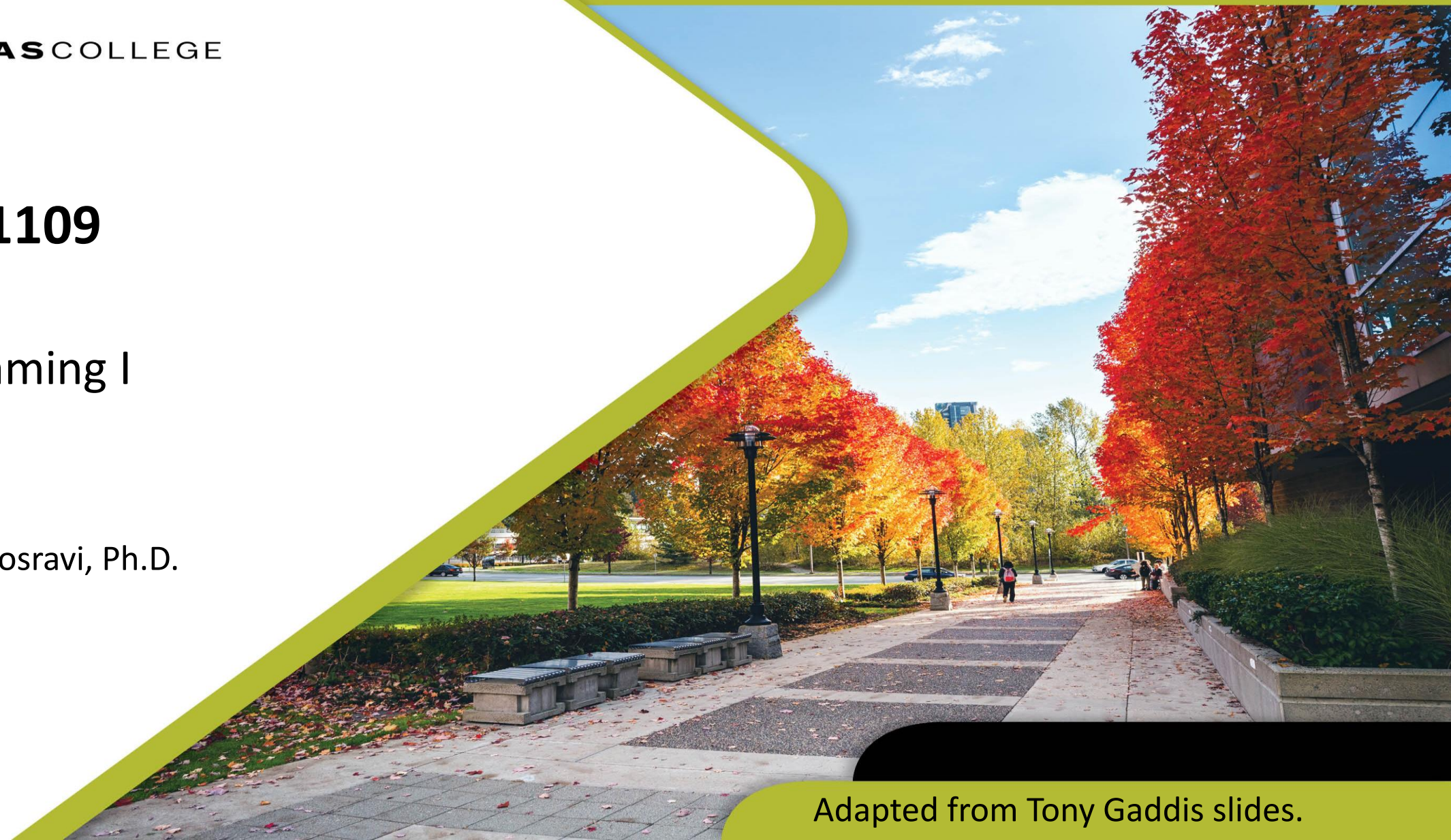
DOUGLAS COLLEGE

CMPT 1109

Programming I

Shahriar Khosravi, Ph.D.

Lecture 6



Adapted from Tony Gaddis slides.

Plan for Today

- Arrays
- Accessing Array Elements
- Bounds Checking
- The Range-Based **`for`** Loop
- Processing Array Contents
- Using Parallel Arrays
- Arrays as Function Arguments
- Two-Dimensional Arrays
- Arrays with Three or More Dimensions
- Introduction to the STL **`vector`**

DOUGLAS COLLEGE

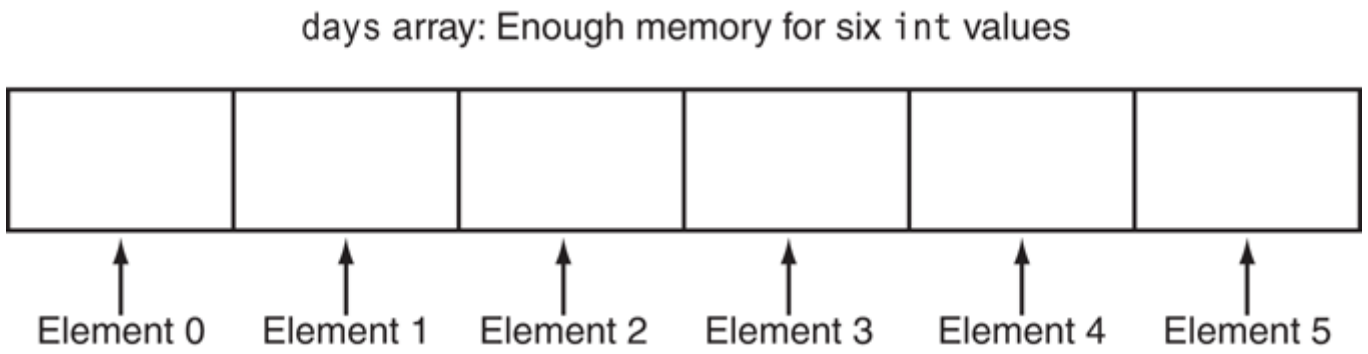
Arrays

Arrays Hold Multiple Values

- **Array**: a variable that can store multiple values of the **same type**.
- Values are stored in adjacent memory locations.
- Arrays are declared using the offset operator `[]`:

```
int tests[5];
```

- The above definition allocates the following memory:



- Note, **index zero is the first element of the array**, so the length of the array here is 6.

Array Terminology

- In the definition **`int tests[5];`**
  - **`int`** is the **data type** of the array elements
  - **`tests`** is the **name** of the array
  - **`5`**, in **`[5]`**, is the **size declarator**. It shows the number of elements in the array.
- The **size** of an array is:
  - the total number of bytes allocated for it.
  - (number of elements) × (number of bytes for each element)

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

Size Declarators

- Arrays of any data type can be defined:

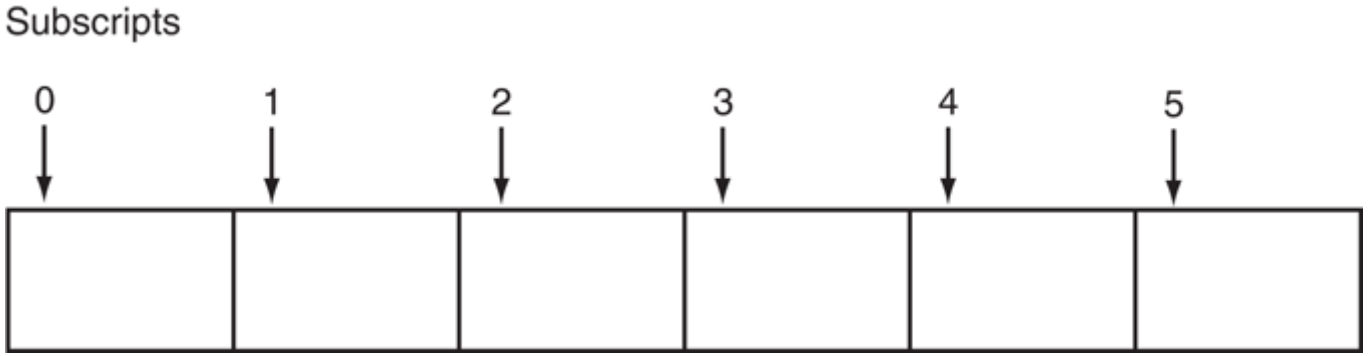
```
float temperatures[100];    // Array of 100 floats
string names[10];           // Array of 10 string objects
long units[50];             // Array of 50 long integers
double sizes[1200];         // Array of 1200 doubles
```

- An array's size declarator must be a constant integer expression with a value greater than zero.
- It can be either a literal, as shown above, or a named constant:

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

Accessing Array Elements

- Each element in an array is assigned a unique subscript (or index).
- Subscripts start at **0**.



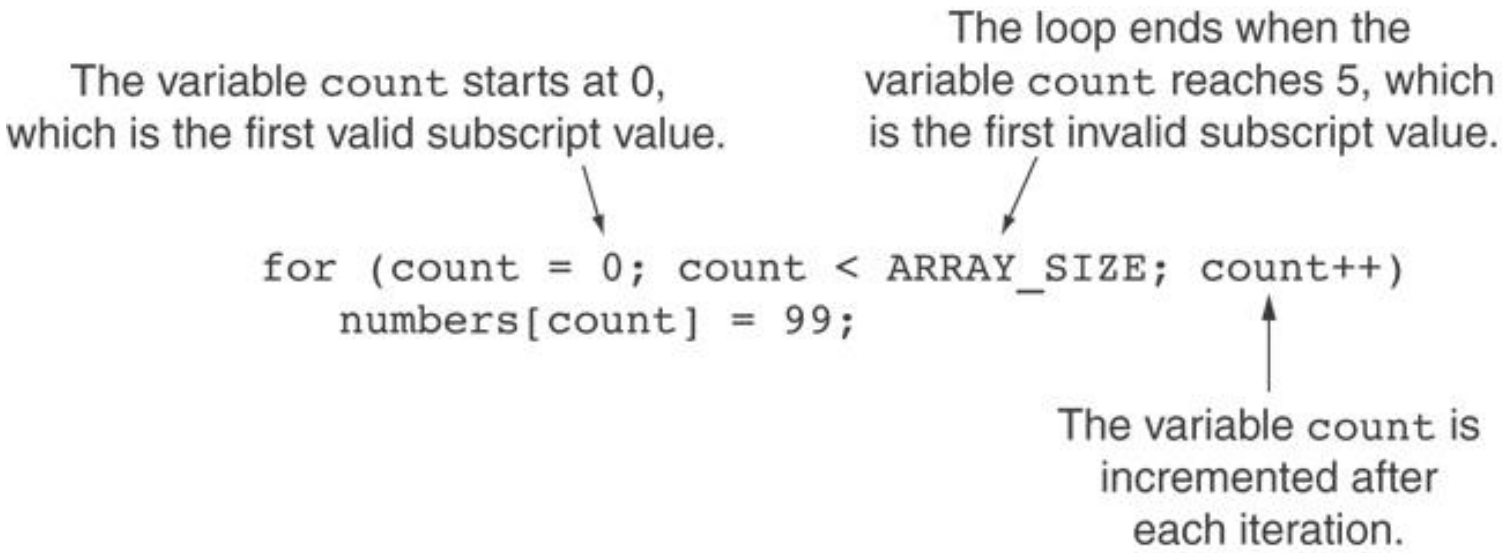
- Array elements can be used as regular variables, and must be accessed individually:

```
int tests[20];
tests[0] = 79;
cout << tests[0];
cin >> tests[1];
tests[4] = tests[0] + tests[1];
```

Using a Loop to Step Through an Array

- Example – The following code defines an array, **numbers**, and assigns **99** to each element:

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

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



Accessing Array Elements

- We can use any integer expression as an array subscript:

```
// 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;                   // Loop counter

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

    // 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;
}
```

Array Initialization

- Arrays can be initialized with an **initialization list**:

```
const int SIZE = 5;
int tests[SIZE] = { 79,82,91,77,84 };
```

- The values are stored in the array in the order in which they appear in the list.
- The initialization list cannot exceed the array size.

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

int main()
{
    const int MONTHS = 12;
    int days[MONTHS] = { 31, 28, 31, 30,
                         31, 30, 31, 31,
                         30, 31, 30, 31 };

    for (int count = 0; count < MONTHS; count++)
    {
        cout << "Month " << (count + 1) << " has ";
        cout << days[count] << " days.\n";
    }
    return 0;
}
```

Array Initialization with Strings

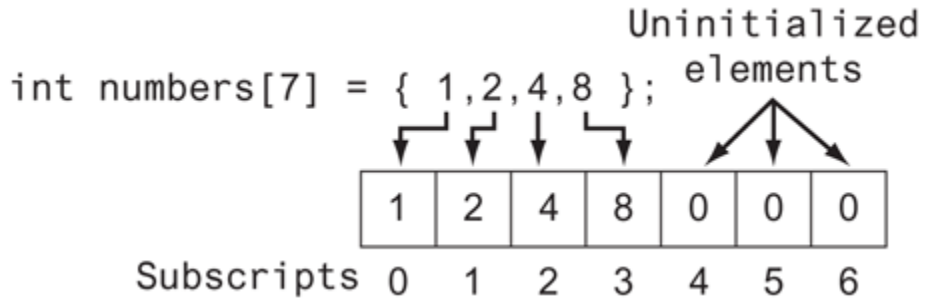
```
// This program initializes a string array.
#include<iostream>
#include<string>
using namespace std;

int main()
{
    const int SIZE = 9;
    string planets[SIZE] = { "Mercury", "Venus", "Earth", "Mars",
                             "Jupiter", "Saturn", "Uranus",
                             "Neptune", "Pluto (a dwarf planet)" };

    cout << "Here are the planets:\n";

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

Array Initialization with Characters

```
// This program uses an array of ten characters to store the
// first ten letters of the alphabet. The ASCII codes of the
// characters are displayed.
#include <iostream>
using namespace std;

int main()
{
    const int NUM_LETTERS = 10;
    char letters[NUM_LETTERS] = { 'A', 'B', 'C', 'D', 'E',
                                  'F', 'G', 'H', 'I', 'J' };

    cout << "Character" << "\t" << "ASCII Code\n";
    cout << "---------" << "\t" << "----------\n";
    for (int count = 0; count < NUM_LETTERS; count++)
    {
        cout << letters[count] << "\t\t";
        cout << static_cast<int>(letters[count]) << endl;
    }
    return 0;
}
```

Partial Array Initialization

- If an array is initialized with fewer initial values than the size declarator, the remaining elements will be set to **0**:



- The uninitialized elements of a string array will contain empty strings.
- If a local array is completely uninitialized, its elements will contain "garbage," like all other local variables.
- If we leave an element uninitialized, we must leave all the elements that follow it uninitialized as well!

```
int numbers[6] = { 2, 4, , 8, , 12 }; // NOT Legal!
```

Implicit Array Sizing

- It is possible to define an array without specifying its size, as long as we provide an initialization list.
- C++ automatically makes the array large enough to hold all the initialization values.
- For example, the following definition creates an array with five elements:

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

- We must provide an initialization list if we leave out an array's size declarator. Otherwise, the compiler doesn't know how large to make the array.

In-Class Exercise

Write a C++ program that reads integer data from a file into an array, then increments each element of the array by one, and writes the modified elements of the array to a separate output file.

In-Class Exercise

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

int main()
{
    const int ARRAY_SIZE = 10; // Array size
    int numbers[ARRAY_SIZE];   // Array with 10 elements
    int count = 0;             // Loop counter variable
    ifstream inputFile;        // Input file stream object
    ofstream outputFile;       // Output file stream object

    // Open the input file.
    inputFile.open("TenNumbers.txt");

    if (!inputFile)
    {
        cout << "ERROR: input file did not open correctly. Terminating..." << endl;
        exit(EXIT_FAILURE);
    }

    // Read the numbers from the file into the array.
    while (count < ARRAY_SIZE)
    {
        inputFile >> numbers[count];
        numbers[count]++;
        count++;
    }

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

    // Open the input file.
    outputFile.open("TenNumbers++.txt");

    if (!outputFile)
    {
        cout << "ERROR: output file did not open correctly. Terminating..." << endl;
        exit(EXIT_FAILURE);
    }

    // Write the numbers out to file:
    for (count = 0; count < ARRAY_SIZE; count++)
        outputFile << numbers[count] << endl;

    outputFile << endl;
    outputFile.close();

    return 0;
}
```

DOUGLAS COLLEGE

Bounds Checking

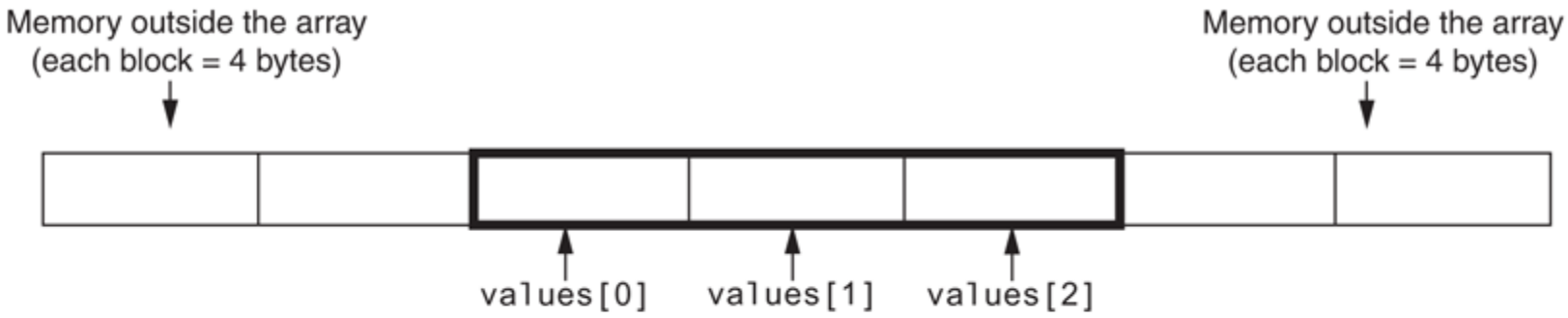
No Array Bounds Checking in C++

- C++ is a popular language for software developers who have to write fast, efficient code.
- To increase runtime efficiency, C++ does **not** provide many of the common safeguards to prevent unsafe memory access found in other languages.
- 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, so be very careful!

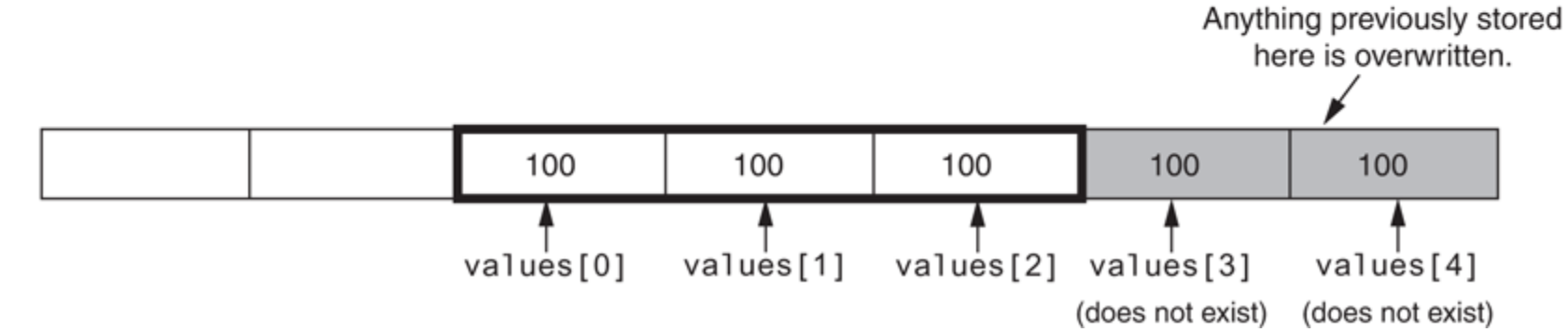
```
// This program unsafely accesses an area of memory by writing
// values beyond an array's boundary.
// WARNING: If you compile and run this program, it could crash.
#include <iostream>
using namespace std;

int main()
{
    const int SIZE = 3;  // Constant for the array size
    int values[SIZE];    // An array of 3 integers
    int count;           // Loop counter variable

    // Attempt to store five numbers in the three-element array.
    cout << "I will store 5 numbers in a 3 element array!\n";
    for (count = 0; count < 5; count++)
        values[count] = 100;

    // If the program is still running, display the numbers.
    cout << "If you see this message, it means the program\n";
    cout << "has not crashed! Here are the numbers:\n";
    for (count = 0; count < 5; count++)
        cout << values[count] << endl;
    return 0;
}
```

What the Code Does

The way the `values` array is set up in memory.
The outlined area represents the array.



How the numbers assigned to the array overflow the array's boundaries.
The shaded area is the section of memory illegally written to.



DOUGLAS COLLEGE

Range-Based `for` Loop

The Range-Based for Loop

- C++ 11 provides a specialized version of the **for** loop that, in many circumstances, simplifies array processing.
- The **range-based for loop** is a loop that iterates once for each element in an array.
- Each time the loop iterates, **it copies an element from the array to a built-in variable**, known as **the range variable**.
- The range-based for loop automatically knows the number of elements in an array.
  - You do not have to use a counter variable (Yaaaay).
  - You do not have to worry about stepping outside the bounds of the array (Yaaaay).

```
for (dataType rangeVariable : array)
    statement;
```

  - **dataType** is the data type of the range variable.
  - **rangeVariable** is the name of the range variable, receives the value of a different array element during each loop iteration.
  - **array** is the name of an array on which we wish the loop to operate.
  - **statement** is a statement that executes during a loop iteration. If you need to execute more than one statement in the loop, enclose within **{ }**

Example

```
// This program demonstrates the range-based for loop.
#include <iostream>
using namespace std;

int main()
{
    // Define an array of integers.
    int numbers[] = { 10, 20, 30, 40, 50 };

    // Display the values in the array.
    for (auto val : numbers)
        cout << val << endl;

    return 0;
}
```

Example

```
// This program demonstrates the range-based for loop.
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string planets[] = { "Mercury", "Venus", "Earth", "Mars",
                         "Jupiter", "Saturn", "Uranus",
                         "Neptune", "Pluto (a dwarf planet)" };

    cout << "Here are the planets:\n";

    // Display the values in the array.
    for (string val : planets)
        cout << val << endl;

    return 0;
}
```

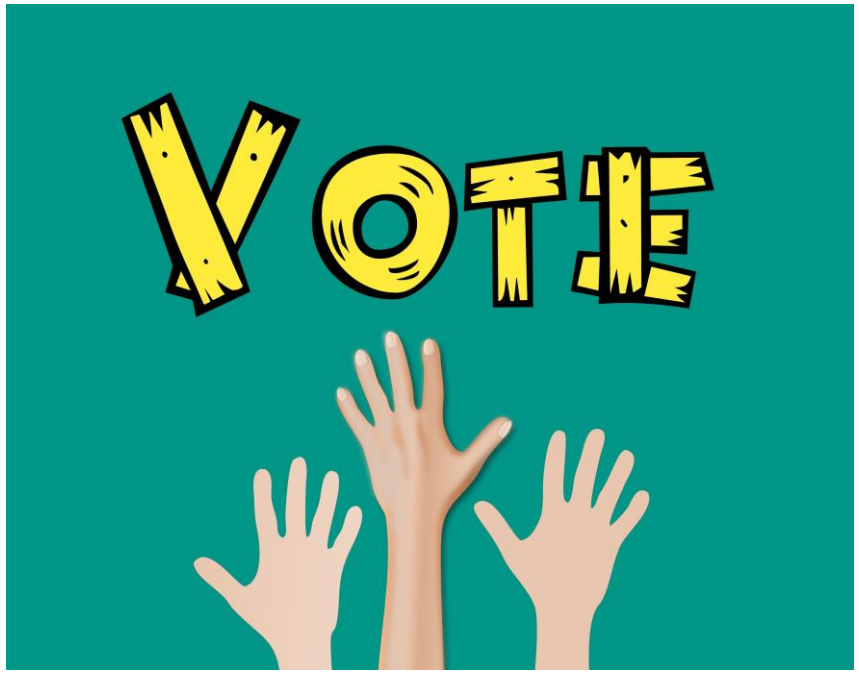
Poll 1 (Extra Credit)

In a range-based for loop, the range variable can be used to modify the element values of an array. For example,

```
for (string val : planets)
        val = "Sun";
```

will set the contents of all elements of `planets` **to** `"sun"`**.**

a) Yay!
b) Nay!



Please use the “Poll” window to participate for extra credit! One answer only please!

Modifying an Array with a Range-Based for Loop

- In a range-based **for** loop, the **range variable contains only a copy of an array element**.
- We **cannot** use a range-based for loop to modify the contents of an array **unless we declare the range variable as a reference**.
- To declare the range variable as a reference variable, simply write an ampersand (**&**) in front of its name in the loop header.
- The range-based **for** loop can be used in any situation where we need to step through the elements of an array, and we do **not** need to use the element subscripts.
- If we need the element subscript for some purpose, we must use the regular **for** loop.

```
// This program uses a range-based for loop to
// modify the contents of an array.
#include <iostream>
using namespace std;

int main()
{
    const int SIZE = 5;
    int numbers[5];

    // Get values for the array.
    for (int& val : numbers)
    {
        cout << "Enter an integer value: ";
        cin >> val;
    }

    // Display the values in the array.
    cout << "Here are the values you entered:\n";
    for (auto val : numbers)
        cout << val << endl;

    return 0;
}
```

DOUGLAS COLLEGE

Processing Array Contents

Processing Array Contents

- Array elements can be treated as ordinary variables of the same type as the array.
- When using **++**, **--** operators, do **not** confuse the element with the subscript:

```
tests[i]++; // add 1 to tests[i]
tests[i++]; // increment i, no
            // effect on tests
```

- To copy one array to another, do **not** try to assign one array to the other:

```
newTests = tests;  // Won't work
```

- Instead, assign element-by-element:

```
for (i = 0; i < ARRAY_SIZE; i++)
        newTests[i] = tests[i];
```

Printing the Contents of an Array

- We can display the contents of a **char** array by sending its name to **cout**:

```
char fName[] = "Henry";
cout << fName << endl;
```

- **However, this only works for character arrays!**
- For other types of arrays, we must print element-by-element:

```
for (i = 0; i < ARRAY_SIZE; i++)
        cout << tests[i] << endl;
```

- Since C++ 11, we can use the range-based **for** loop to display an array's contents:

```
for (int val : tests)
        cout << val << endl;
```

Summing and Averaging Array Elements

- Use a simple loop to add together array elements:

```
double total = 0;   // Initialize accumulator
double average;     // Will hold the average
for (int count = 0; count < NUM_SCORES; count++)
    total += scores[count];
average = total / NUM_SCORES;
```

- Or use a range-based **for** loop:

```
double total = 0;  // Initialize accumulator
double average;    // Will hold the average
for (int val : scores)
    total += val;
average = total / NUM_SCORES;
```

Finding the Highest and Lowest Values in a Numeric Array

- When this code is finished, the **highest** variable will contains the highest value in the **numbers** array.

```
const int SIZE = 50;
int numbers[SIZE];
int count;
int highest;

highest = numbers[0];
for (count = 1; count < SIZE; count++)
{
    if (numbers[count] > highest)
        highest = numbers[count];
}
```

- When this code is finished, the **lowest** variable will contains the lowest value in the **numbers** array.

```
const int SIZE = 50;
int numbers[SIZE];

int count;
int lowest;
lowest = numbers[0];
for (count = 1; count < SIZE; count++)
{
    if (numbers[count] < lowest)
        lowest = numbers[count];
}
```

Partially-Filled Arrays

- If it is unknown how much data an array will be holding:
  - Make the array large enough to hold the largest expected number of elements.
  - Use a counter variable to keep track of the number of items stored in the array.

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

int main()
{
    const int SIZE = 100;
    int numbers[SIZE];
    int count = 0;

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

    for (int index = 0; index < count; index++)
    {
        cout << numbers[index] << endl;
    }

    return 0;
}
```

Comparing Arrays

- To compare two arrays, you must compare element-by-element:

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

int main()
{
    const int SIZE = 5;
    int firstArray[SIZE] = { 5, 10, 15, 20, 25 };
    int secondArray[SIZE] = { 5, 10, 15, 20, 25 };
    bool arraysEqual = true; // Flag variable
    int count = 0;           // Loop counter variable
    // Compare the two arrays.
    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;
}
```

Using Parallel Arrays

- **Parallel arrays**: two or more arrays that contain related data.
- A subscript is used to relate arrays: elements at same subscript are related.
- Arrays may be of different types.

```
const int SIZE = 5;    // Array size
int id[SIZE];          // student ID
double average[SIZE]; // course average
char grade[SIZE];      // course grade

...

for (int i = 0; i < SIZE; i++) {
    cout << "Student ID: " << id[i]
         << " average: " << average[i]
         << " grade: " << grade[i]
         << endl;
}
```

DOUGLAS COLLEGE

Arrays and Functions

Arrays as Function Arguments

- To pass an array to a function, just use the array name:

```
showScores(tests);
```

- To define a function that takes an array parameter, use empty [ ] for array argument:

```
// function prototype
void showScores(int[]);
// function header
void showScores(int tests[])
```

- When passing an array to a function, it is common to pass array size so that function knows how many elements to process:

```
showScores(tests, ARRAY_SIZE);
```

- Array size must also be reflected in prototype, header:

```
// function prototype
void showScores(int[], int);
// function header
void showScores(int tests[], int size)
```

Example

```
// This program demonstrates an array being passed to a function.
#include <iostream>
using namespace std;

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;
}
```

Modifying Arrays in Functions

- Array names in functions are like reference variables – **changes made to array in a function are reflected in actual array in calling function**.
- We must exercise caution that an array is not inadvertently changed by a function.
- We can prevent a function from making changes to an array argument by using the **`const`** keyword in the parameter declaration.

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

```
// This program uses a function to double the value of
// each element of an array.
#include <iostream>
using namespace std;

// Function prototypes
void doubleArray(int[], int);
void showValues(int[], int);

int main()
{
    const int ARRAY_SIZE = 7;
    int set[ARRAY_SIZE] = { 1, 2, 3, 4, 5, 6, 7 };

    // Display the initial values.
    cout << "The arrays values are:\n";
    showValues(set, ARRAY_SIZE);

    // Double the values in the array.
    doubleArray(set, ARRAY_SIZE);

    // Display the resulting values.
    cout << "After calling doubleArray the values are:\n";
    showValues(set, ARRAY_SIZE);

    return 0;
}

void doubleArray(int nums[], int size)
{
    for (int index = 0; index < size; index++)
        nums[index] *= 2;
}

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

Poll 2 (Extra Credit)

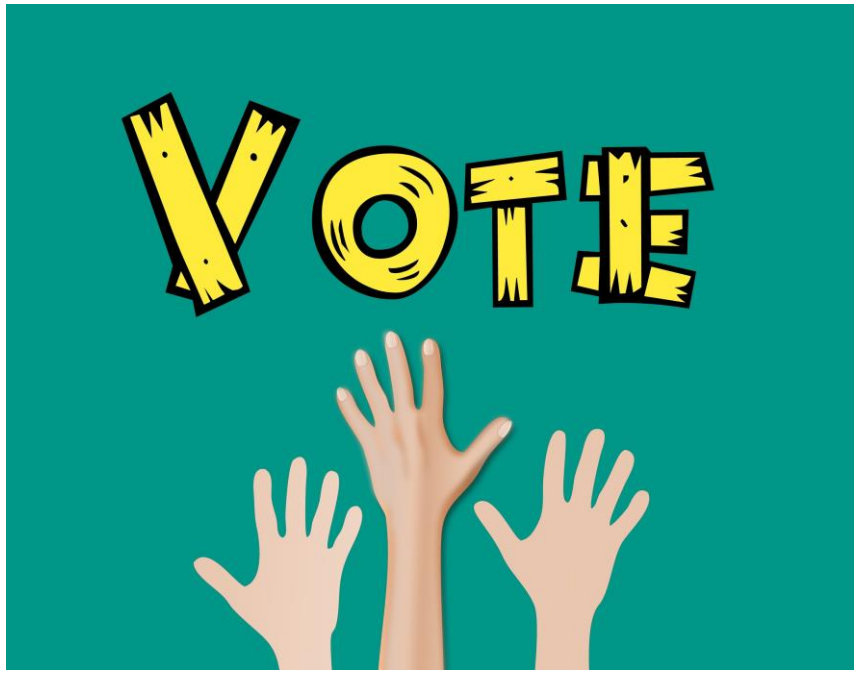
Given the following array definitions:

```
double array1[4] = { 1.2, 3.2, 4.2, 5.2 };
double array2[4];
```

will the following statement work?

```
array1 = array2
```

a) Yay!
b) Nay!



Please use the "Poll" window to participate for extra credit! One answer only please!

DOUGLAS COLLEGE

Multi-Dimensional Arrays

Two-Dimensional Arrays

- A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data (e.g., student IDs in rows and scores in columns).
- To define a 2D array, use two size declarators in definition:

```
const int ROWS = 4, COLS = 3;
int exams[ROWS][COLS];
```

- First declarator is number of rows; second is number of columns.
- This declaration creates the following array in memory:

columns

rows

exams[0][0]	exams[0][1]	exams[0][2]
exams[1][0]	exams[1][1]	exams[1][2]
exams[2][0]	exams[2][1]	exams[2][2]
exams[3][0]	exams[3][1]	exams[3][2]

Use two subscripts to access an element:

```
exams[0][1];
```

Example

```
// 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 divisions.\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;
}
```

2D Array Initialization

- Two-dimensional arrays are initialized row-by-row:

```
const int ROWS = 2, COLS = 2;int exams[ROWS][COLS] = { {84, 78},
                                                        {92, 97} };
```

84	78
92	97

- We can omit the inner **{ }**, but it is a good idea to keep it for clarity.
- The braces also give us the ability to leave out initializers within a row without omitting the initializers for the rows that follow it. For instance, in the following initialization

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

- **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 initialized to zero automatically.

Passing Two-Dimensional Arrays to Functions

- Use array name as argument in function call:

```
getExams(exams, 2);
```

- When a two-dimensional array is passed to a function, the parameter type must contain a size declarator for the number of columns.

```
const int COLS = 2;
// Prototype
void getExams(int [][COLS], int);
// Header
void getExams(int exams[][COLS], int rows)
```

- The function can accept any two-dimensional integer array, as long as it consists of four columns.
- C++ requires the columns to be specified in the function prototype and header because of the way two-dimensional arrays are stored in memory. One row follows another

Example

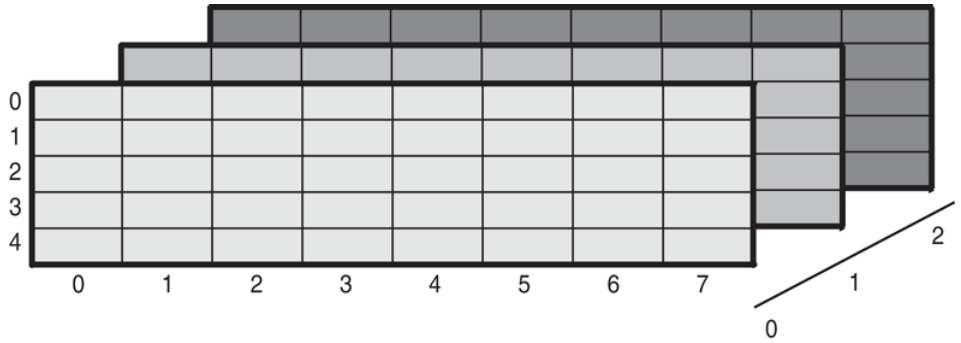
```
// This program demonstrates accepting a 2D array argument.
#include <iostream>
#include <iomanip>
using namespace std;

// 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(const 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} };

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

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.

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

- This array can be thought of as three sets of five rows, with each row containing eight elements.



- Arrays with more than three dimensions are difficult to visualize, but can be useful in some programming problems.
- When writing functions that accept multi-dimensional arrays as arguments, all but the first dimension must be explicitly stated in the parameter list.

Thank you.

DOUGLAS COLLEGE

DOUGLAS COLLEGE