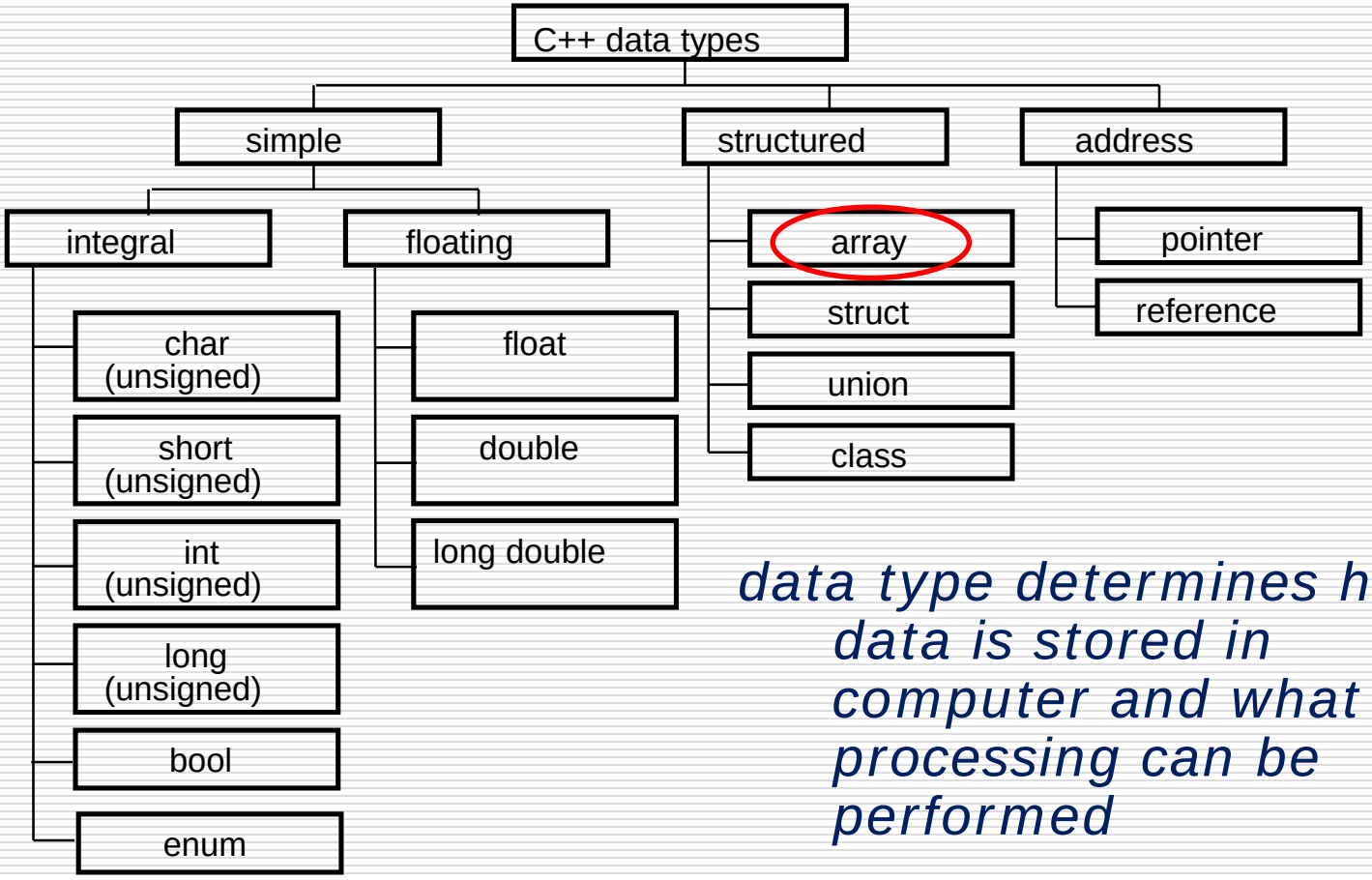
Ch7: Arrays

- Array Data Types
  - Array Elements
  - Array Initialization and Assignment
  - Bounds Checking
  - C++ 11 Range-Based for Loop
  - Arrays as Function Parameters
- Files and Arrays
- Array Operations
- Parallel Arrays
- Multi-Dimensional Arrays

Data Types

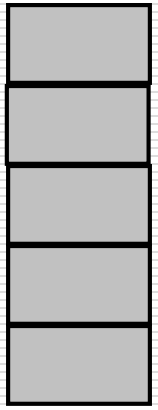


data type determines how data is stored in computer and what processing can be performed

Array Data Types

- Array is a sequence of data items that are
  - of the same type (homogeneous)
  - accessed through an integer index
  - stored contiguously in memory, starting at base address
    - array size must be constant integer expression greater than 0
    - amount of memory used is equal to number of elements times the number of bytes for each element

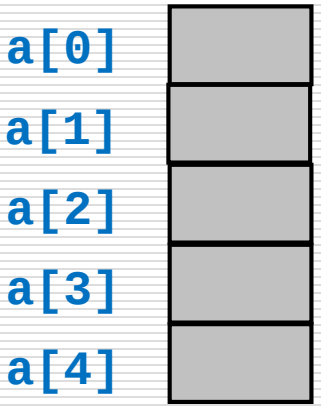
```
int a[5];
```



Array Elements

- an element is a component of an array
- each element has an associated index number (starting at 0)
- declaration has the following form
  - type name[size];
- integer indexing/subscripting to reference individual elements of array (be careful)
  - lowerbound index = 0
  - upperbound index = size - 1
  - size = upperbound index + 1

```
int a[5];
```



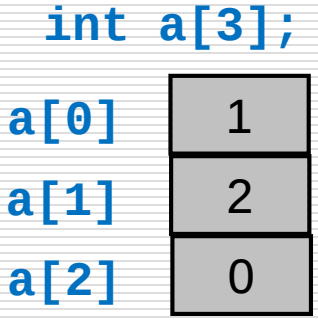
Array Initialization

- initialization has the following form
  - type name[size] = { init-list };
    - size must be constant integer expression with value greater than 0
- explicitly
  - upon declaration
    - sized array

```
int a[3] = {1, 2};
```

    - unsized array
      - size determined from initialization list

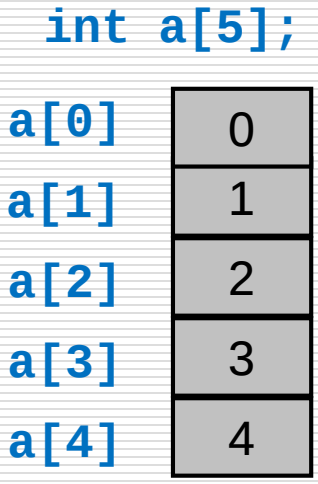
```
int a[] = {1, 2, 0};
```



Array Initialization

- explicitly
  - after declaration and within program
    - looping through elements

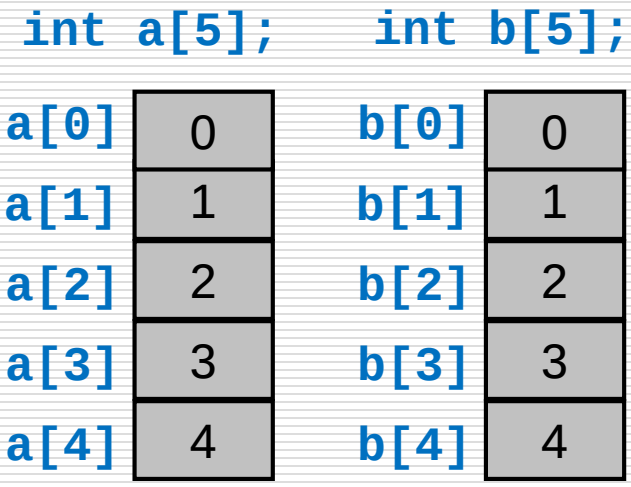
```
int i, a[5];
for(i = 0; i < 5; ++i)
    a[i] = i;
```



Array Initialization vs Array Assignment

- array elements are accessed individually for assignment (and other operations)
- example

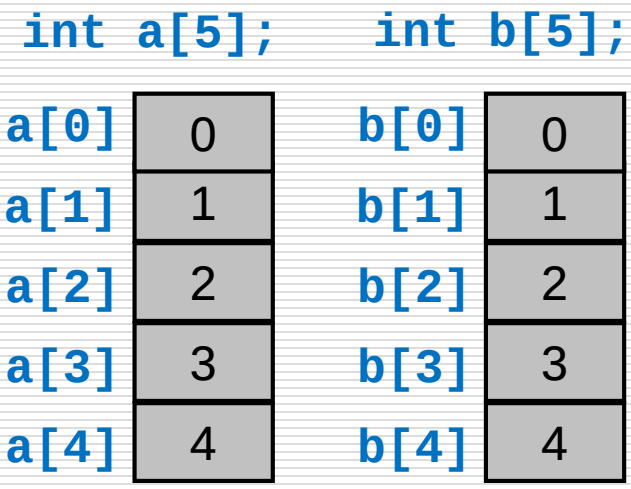
```
int i, a[5], b[5];
for (i = 0; i < 5; ++i)
   a[i] = i;
for (i = 4; i >= 0; --i)
   b[i] = i;
```



Array Initialization vs Array Assignment

- arrays cannot be assigned to other arrays
  - elements are copied
- example

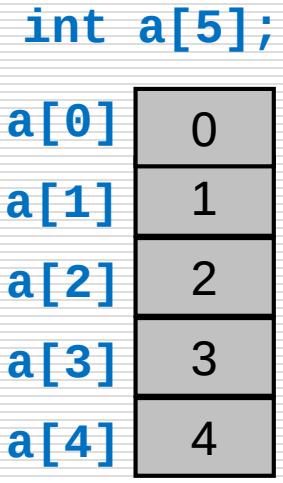
```
int i, a[5], b[5];
for (i = 0; i < 5; ++i)
   a[i] = i;
b = a; // WRONG!
for (i = 0; i < 5; ++i)
   b[i]= a[i];
```



Array Input And Output

- array elements are accessed individually for input or output
- example

```
int i, a[5];
for (i = 0; i < 5; ++i)
   cin >> a[i];
for (i = 0; i < 5; ++i)
   cout << a[i];
```



- what is difference between
  - a[i++] ← **increment i**
  - a[i]++ ← **increment value stored at a[i]**

Array Bounds Checking

- C++ does not provide bound checking for arrays

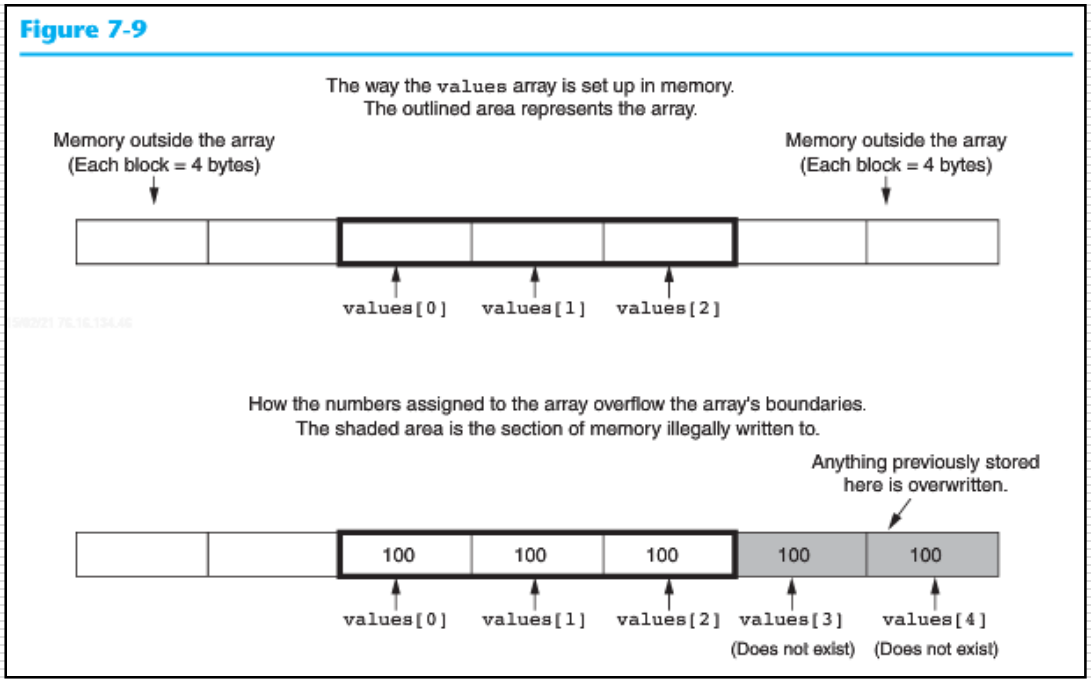
```
const int SIZE = 3;
int values[SIZE];
int count;

// trying to store more data than elements
for (count = 0; count < 5; count++)
   values[count] = 100;

for (count = 0; count < 5; count++)
   cout << values[count] << endl;
```

Array Bounds Checking

- C++ does not provide bound checking for arrays



Array Bounds Checking

- watch for off-by-one errors

```
const int SIZE = 3;
int values[SIZE];
int count;

// trying to store more data than elements
for (count = 0; count <= SIZE; count++)
   values[count] = 100;
```

Range-Based for Loop

- Introduced with C++11
  - Loop automatically iterates once for each element in array
  - Range variable has copy of array element

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

```
int numbers[] = {3, 6, 9};
for (int val : numbers)
   cout << val << endl;
```

Range-Based for Loop

- Can modify with reference range variable

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

```
int numbers[5];
for (int &val : numbers)
   val = 100;
```

Arrays as Function Arguments

- Array parameters include brackets and array size
  - By default array is reference whose values can change
  - Use const keyword to prevent modification of array elements
- Only use array name in function call
  - Do not include brackets in function call

Arrays as Function Arguments

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

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

Arrays as Function Arguments

```
const int SIZE = 7;
int set[SIZE] = {1, 2, 3, 4, 5, 6, 7};
cout << “The array’s values are:\n”;
showValues(set, SIZE);
doubleArray(set, SIZE);
cout << “After calling doubleArray the values are:\n”;
showValues(set, SIZE);
```

```
The array’s values are:
1 2 3 4 5 6 7
After calling doubleArray the values are:
2 4 6 8 10 12 14
```

Reading Data From File Into Array

- reading data into array continues until
  - array is filled, OR
  - end of file is reached

```
ifstream inFile("TenNumbers.txt");
const int ARR_SIZE = 10;
float numbers[ARR_SIZE];
int count = 0;
...// check for file open errors
while (count < ARR_SIZE && inFile >> numbers[count])
   count++;
inFile.close();
```

Reading Data From File Into Array

- reading data into array *(another version)*

```
ifstream inFile("TenNumbers.txt");
const int ARR_SIZE = 10;
float numbers[ARR_SIZE];
int count = 0;
...// check for file open errors
while (count < ARR_SIZE && !inFile.eof()){
   inFile >> numbers[count];
   count++;
}
inFile.close();
```

Writing Data From Array To File

- writing data from array into file

```
ifstream outFile("SaveNumbers.txt");
const int ARR_SIZE = 10;
float numbers[ARR_SIZE];
int count;
...// check for file open errors
...// populate array with data
// output all populated array elements
for (count = 0; count < ARR_SIZE; count++)
   outFile << numbers[count] << endl;
outFile.close();
```

Array Operations: Summing Values in an Array

- use a loop with an accumulator

```
const int NUM_UNITS = 24;
int units[NUM_UNITS];
int total = 0;  // initialize accumulator

...// populate array with data

for (int count = 0; count < NUM_UNITS; count++)
   total += units[count];

for (int val : units) // C++ 11 range-based for
   total += val;
```

Array Operations: Calculating Average

- sum values of all elements
- divide sum by number of elements

```
const int NUM_SCORES = 10;
double scores[NUM_SCORES];
double total = 0;  // initialize accumulator
double average;    // will hold average
...// populate array with data
for (int count = 0; count < NUM_SCORES; count++)
   total += scores[count];

average = total / NUM_SCORES;
```

Array Operations: Finding Highest/Lowest Value

- initialize highest/lowest to first element
- compare remaining elements with highest/lowest

```
const int SIZE = 50;
double numbers[SIZE];
...// populate array with data
// initialize highest value
double highest = numbers[0];
// compare remaining elements
for (int count = 1; count < SIZE; count++)
   if (numbers[count] > highest)
      highest = numbers[count];
```

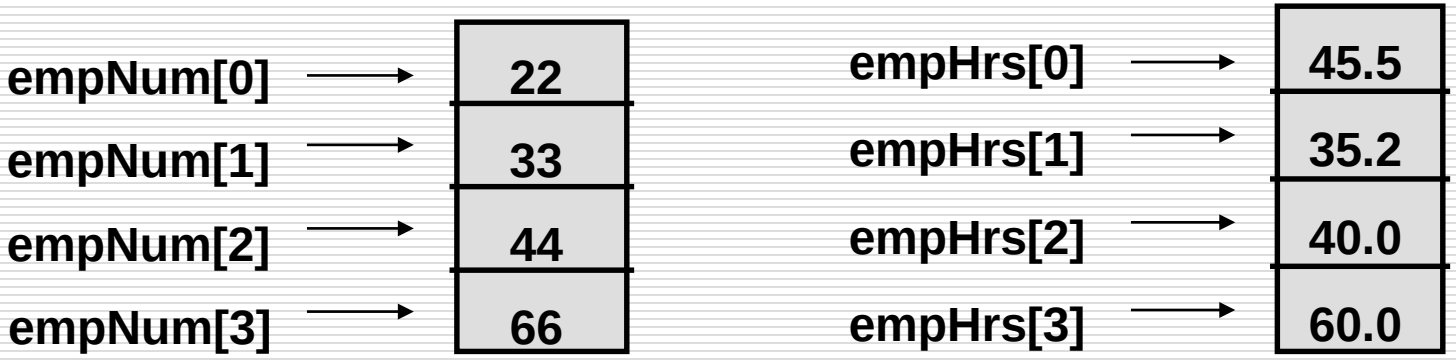
Array Operations: Partially Filled Arrays

- exact number of elements *unknown*
- create array to hold largest possible number of items

```
const int SIZE = 100;
double numbers[SIZE];
int count = 0;
ifstream inFile("Numbers.txt");
...// check for file open errors
while (count < SIZE && !inFile.eof())
   inFile >> numbers[count++];
cout << "Number of items in file "
   << count << endl;
inFile.close();
```

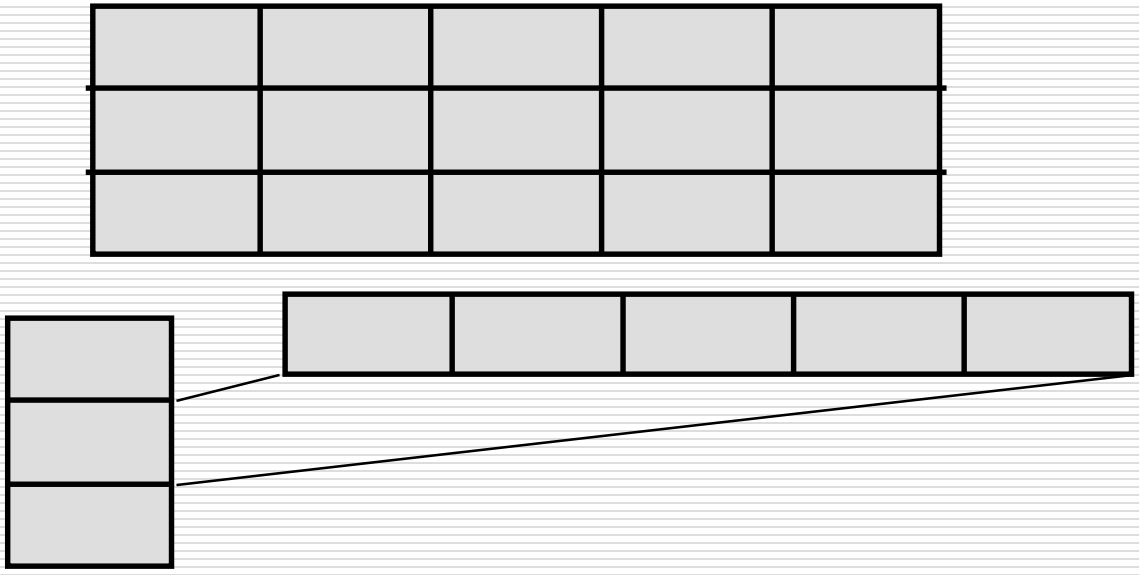
Parallel Arrays

- two or more arrays have a relationship between data stored in relative positions
  - int empNum[10];
  - float empHrs[10];



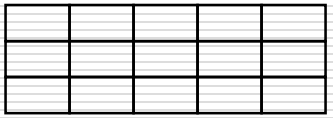
Multi-Dimensional Arrays

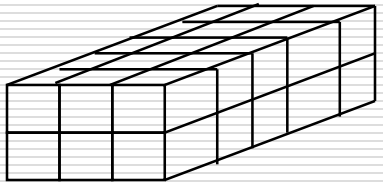
- an n-dimensional array can be seen as a one-dimensional array, each of who elements is itself an [n - 1] dimensional array



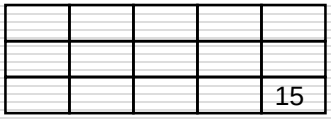
Multi-Dimensional Arrays

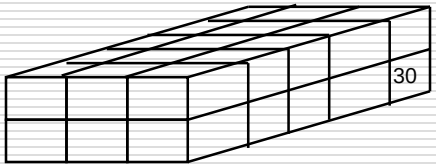
- *each* pair of brackets in array definition adds dimension
  - int a[3][5];
  - int a[2][3][5];





- to access *specific* element
  - integer index or subscript is needed for *each* dimension
  - a[2][4] = 15;
  - a[1][2][4] = 30;





Multi-Dimensional Arrays

- to initialize multi-dimensional array
  - use multiple initializer lists
  - 2-D example
    - int a[3][5] ={1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
      11, 12, 13, 14, 15};
    - int a[3][5] = {{1, 2, 3, 4, 5},
      {6, 7, 8, 9, 10},
      {11, 12, 13, 14, 15}};
    - int a[][5] = {{1, 2, 3, 4, 5},
      {6, 7, 8, 9, 10},
      {11, 12, 13, 14, 15}};

Multi-Dimensional Arrays

- 3-D example
  - int a[2][3][5] = {{{1, 1, 0, 0, 0}, {2, 0, 0, 0, 0},
    {3, 3, 0, 0, 0}},
    {{4, 4, 0, 0, 0}, {4, 0, 0, 0, 0},
    {5, 0, 0, 0, 0}}};
  - int a[][3][5] = {{{1, 1}, {2}, {3, 3}},
    {{4, 4}, {4}, {5}}};

- to access *all* values of multi-dimensional array
  - need loop for *each* dimension
    - value assignment
    - value summation

Multi-Dimensional Arrays as Parameters

- to declare parameter for multi-dimensional array
  - *must* specify number of cells in all dimensions beyond the first (arrays stored in row order)
  - 2-D example
    - int sum(int a[3][5], int size);
    - int sum(int a[][5], int size);
  - 3-D example
    - int sum(int a[2][3][5], int size);
    - int sum(int a[][3][5], int size);

Multi-Dimensional Arrays as Parameters

- 2-D example (sum *all* elements):

```
int sum(int a[][5], int n)
{
  int i, j, sumVal = 0;

  for (i = 0; i < n; ++i)
    for (j = 0; j < 5; ++j)
      sumVal += a[i][j];

  return sumVal;
}
```

Loop for each array dimension.

```
int sumArr, iArr[3][5];
sumArr = sum(iArr, 3);
cout << sumArr;
```

Multi-Dimensional Arrays as Parameters

- 3-D example (sum *all* elements):

```
int sum(int a[][3][5], int n)
{
  int i, j, k, sumVal = 0;

  for (i = 0; i < n; ++i)
    for (j = 0; j < 3; ++j)
      for (k = 0; k < 5; ++k)
        sumVal += a[i][j][k];

  return sumVal;
}
```

Loop for each array dimension.

```
int sumArr, iArr[2][3][5];
sumArr = sum(iArr, 2);
cout << sumArr;
```

Summing Rows of 2D Array

```
for (int row = 0; row < MAX_ROW; row++)
{
  rowTotal = 0;

  for (int col = 0; col < MAX_COL; col++)
    rowTotal += scores[row][col];

  cout << "Row " << (row + 1) << " total = "
    << rowTotal << endl;
}
```

Inner loop holds row constant and calculates sum for all columns in row.

Summing Columns of 2D Array

```
for (int col = 0; col < MAX_COL; col++)
{
  colTotal = 0;

  for (int row = 0; row < MAX_ROW; row++)
    colTotal += scores[row][col];

  cout << "Col " << (col + 1) << " total = "
    << colTotal << endl;
}
```

Inner loop holds column constant and calculates sum for all rows in column.