COP 3014 – Programming I

Chapter 6 – Arrays

Simple & Structured Data Types

- Simple Data Types
  - Can only store one value at a time
  - Example of Types:
    - integral (char, short, int, long, and bool)
    - floating (float, double, long double)
- Structured Data Types
  - Each data item is a collection of other data items
  - Used to group related data of various types for convenient access using the same identifier
  - user-defined data type (UDT)
  - Example of Types:
    - array
    - Struct
    - class

Arrays – Description

- Array: collection of fixed number of components (elements), wherein all of components have **same data type**
- One-dimensional array: array in which components are arranged in list form
- Multi-dimensional array: array in which components are arranged in tabular form (more info in text but not covered in class)
- *Array Basics:* Consecutive group of memory locations that all have the **same type**
- The collection of data is indexed, or numbered, and at starts at 0
- Position number is formally called the *subscript* or *index*
  - First element is subscript 0 (zero), sometimes called the zeroth element.
  - The highest element index is one less than the total number of elements in the array

Arrays – Visual Layout

- Example of array named ‘c’ with 12 elements already initialized to the values shown:

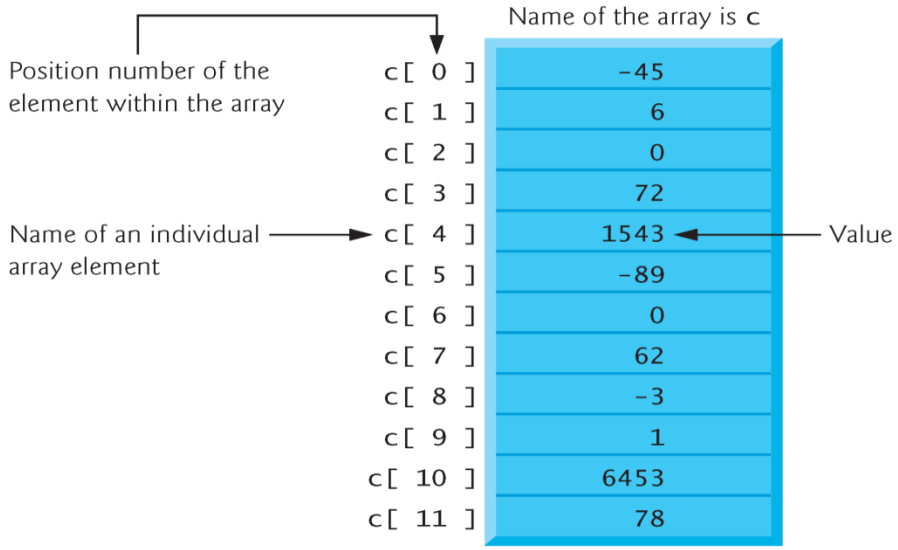

Fig 6.1 Array of 12 elements.

Arrays – Basics

- *Array Basics:* Consecutive group of memory locations that all have the **same type**
- The collection of data is indexed, or numbered, and at starts at 0
- Position number is formally called the *subscript* or *index*
  - First element is subscript 0 (zero), sometimes called the zeroth element.
  - The highest element index is one less than the total number of elements in the array
- The collection of data is indexed, or numbered, and at starts at 0
- Position number is formally called the *subscript* or *index*
  - First element is subscript 0 (zero), sometimes called the zeroth element.
  - The highest element index is one less than the total number of elements in the array

Arrays – Declaring an Array

- Syntax to declare one-dimensional array:

```
//intExp evaluates to positive integer--indicates number of elements
dataType arrayName[intExp];

//following declares array num containing 5 elements of type int:
//num[0], num[1], num[2], num[3], and num[4]
int num[5];
```

- Index (intExp), any expression whose value is non-negative integer
- intExp indicates number of elements
- Size of array: number of elements
- Compiler reserves the appropriate amount of memory

Arrays – Initialization

- Like simple variables, arrays can be initialized during declaration
- When initializing arrays, not required to specify size of array
- Size of array determined by number of values within braces

```
double sales[] = {12.25, 32.50, 16.90, 23, 45.68};

//same result as...
double sales[5] = {12.25, 32.50, 16.90, 23, 45.68};
```

Arrays – Initialization

- Using a loop; Fig. 6.3: fig06_03.cpp

```
// Initializing an array's elements to zeros and printing the array.
#include <iostream>
#include <iomanip>

int main()
{
  int n[ 10 ]; // n is an array of 10 integers

  // initialize elements of array n to 0
  for ( int i = 0; i < 10; ++i )
    n[ i ] = 0; // set element at location i to 0

  std::cout << "Element" << std::setw( 13 ) << "Value" << std::endl;

  // output each array element's value
  for ( int j = 0; j < 10; ++j )
  std::cout << std::setw( 7 ) << j << std::setw( 13 ) << n[ j ] << std::endl;
} // end main
```

Arrays – Initialization

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

Fig. 6.3 – Initializing an array's elements to zeros and printing the array

Arrays – Initialization

- Initializing with Initialization List can be done only in the declaration
- Using an Initializer List; Fig. 6.4: fig06_04.cpp

```
// Initializing an array in a declaration.
#include <iostream>
#include <iomanip>

int main()
{
  // use initializer list to initialize array n
  int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
  std::cout << "Element" << std::setw( 13 ) << "Value" << std::endl;

  // output each array element's value
  for ( int i = 0; i < 10; ++i )
  std::cout << std::setw( 7 ) << i << std::setw( 13 ) << n[ i ] << std::endl;
} // end main
```

Arrays – Initialization

```
Element         Value
      0            32
      1            27
      2            64
      3            18
      4            95
      5            14
      6            90
      7            70
      8            60
      9            37
```

Fig. 6.4 – Initializing an array in a declaration

Arrays – Initialization

- Specifying Array Size with Constant and Setting Array Elements with Calculations

```
// Fig. 6.5: fig06_05.cpp
// Set array s to the even integers from 2 to 20.
#include <iostream>
#include <iomanip

int main()
{
  // constant variable can be used to specify array size
  const int arraySize = 10; // must initialize in declaration
  int s[ arraySize ]; // array s has 10 elements

  for ( int i = 0; i < arraySize; ++i ) // set the values
    s[ i ] = 2 + 2 * i;

  std::cout << "Element" << std::setw( 13 ) << "Value" << std::endl;

  // output contents of array s in tabular format
  for ( int j = 0; j < arraySize; ++j )
  std::cout << std::setw( 7 ) << j << std::setw( 13 ) << s[ j ] << std::endl;
} // end main
```

Arrays – Initialization

```
Element         Value
      0             2
      1             4
      2             6
      3             8
      4            10
      5            12
      6            14
      7            16
      8            18
      9            20
```

Fig. 6.5 – Generating values to be placed into elements of an array

Arrays – Partial Initialization

- If there are fewer values in the *Initializer List* then the remaining elements are initialized to zero
- If there are no values in the *Initializer List* then all elements are initialized to zero
- Following statement declares array num with 5 elements, and initializes ALL elements to zero (0):

```
int num[5] = {0};
```

- Following statement declares array n with 10 elements and initializes all elements to 0

```
int n[10] = {}; //initialize all elements of array n to 0
```

Arrays – Partial Initialization

- Following statement declares array num with 5 elements, and initializes num[0] to 8, num[1] to 5, num[2] to 12, all others to 0

```
int num[5] = {8, 5, 12};
```

- Following statement declares array num with 3 elements, and initializes num[0] to 5, num[1] to 6, and num[2] to 3

```
int num[] = {5, 6, 3};
```

- Following statement declares array num with 100 elements, and initializes num[0] to 4, num[1] to 7, all others to 0

```
int num[100]= {4, 7};
```

Arrays – Accessing Elements

- Syntax to access array element:

```
//Index value (intExp) specifies position of element in array
arrayName[intExp]

//fifth element in array num
num[4];
```

- Index – any expression whose value is non-negative integer
- Specific index value (e.g., num[0]) indicates position of element in array
- Position number: distance from 1st element of array, beginning at 0 (zero)
- Array subscripting operator []
- Array index always begins at 0

Arrays – Accessing Elements

```
//declare array item with five elements of type int
int item[5];

//assign value 35 to 5th element in item array
item[4] = 35;

//assign value 10 to 4th element in item array
item[3] = 10;

//assign value 45 to 3rd element in item array
item[2] = item[3] + item[4];
```

Arrays – Processing

- Processing One-Dimensional Arrays
- The next few slides will demonstrate the basic array operations using iteration for the following operations:
  1. Initialize
  2. Input
  3. Output
  4. Sum and Average
  5. Find largest element value
  6. Find smallest element value

Arrays – Processing (Setup)

```
//initialize named constant
const int arraySize=5;

//declare array list with arraySize elements of type double
double list[arraySize];

//initialize 7 variables
int i=0; double
smallest=0.0;
double largest=0.0;
double sum=0.0;
double average=0.0;
int maxi=0;
int mini=0;
```

Arrays – Processing

```
//1. initialize each element in array list to 0.0, beginning w/first element
     for (i=0; i < arraySize; ++i)
       list[i]=0.0;


//2. input value for each element in array list, beginning w/first element
     for (i=0; i < arraySize; ++i)
       std:: cin >> list[i];


//3. output value for each element in array list, beginning w/first element
     for (i=0; i < arraySize; ++i)
       std:: cout << list[i] << " ";


//4. sum and average elements in array list, and display
     for (i=0; i < arraySize; ++i)
       sum = sum + list[i];
     average = sum / arraySize;

     std:: cout << "Sum = " << sum;
     std:: cout << "\nAverage = " << average;
```

Arrays – Processing

```
//5. find largest element value in array list, and display
for (i=0; i < arraySize; ++i)
  if (list[maxi] < list[i])
    maxi = i;
  largest = list[maxi];
std::cout << "\nLargest = " << largest;

//6. find smallest element value in array list, and display
for (i=0; i < arraySize; ++i)
  if (list[mini] > list[i])
    mini = i;
  smallest = list[mini];
std:: cout << "\nSmallest = " << smallest;
```

Arrays – Processing Restrictions

- C++ does **not** allow aggregate operations on arrays (e.g., assignment, reading and printing contents of array--must be done element-wise)
- Aggregate operation: any operation that manipulates entire collection (e.g., array) as single unit

```
int myArray[5] = {2, 4, 6, 8, 10};
int yourArray[5];

yourArray = myArray; //illegal assignment
```

Arrays – Processing Restrictions

- Must perform member-wise copy:

```
int myArray[5] = {2, 4, 6, 8, 10};
int yourArray[5];

//legal, member-wise copy
for (int i=0; i < 5; i++)
  yourArray[i] = myArray[i];
```

- Same with IO: must perform member-wise instructions:

```
int myArray[5];

std::cout << myArray; //illegal
std::cin >> myArray; //illegal

//legal, member-wise input
for (int i=0; i < 5; i++)
  std::cin >> myArray[i];

//legal, member-wise output
for (int i=0; i < 5; i++)
  std::cout << myArray[i];
```

Arrays as Function Parameters

- Arrays passed by reference only;
- Must also pass array size to functions
- Ampersand (&) not used when declaring array as formal parameter--still passed by reference
- Base address of array passed as formal parameter
  - Array name is address of the first element
- Functions can modify element values
- Good programming practice: using reserved word `const` in declaration of array as formal parameter (when needed)
  - prevents function from altering actual parameter

```
//Function to print array:
//array and number of elements passed to parameters
//listSize specifies number of elements to be printed
void printArray(const int list[], int listSize)
{
  int counter;

  for (counter = 0; counter < listSize; counter++)
    std::cout << list[counter] << " ";
}
```

Arrays as Parameters

- In function call statement, when passing array as actual parameter, only use array name--NO brackets ([]):

```
//Call to function printArray(), list is one-dimensional array
//arraySize is scalar variable of type integer
printArray(list, arraySize);
```

```
// Fig. 6.13: fig06_13.cpp
// Passing arrays and individual array elements to functions.
#include <iostream>
#include <iomanip>
                    // ↓ can include the name but not required
void modifyArray( int [], int ); // appears strange; array and size
void modifyElement( int ); // receive array element value

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

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

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

   std::cout << endl;
```

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

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

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

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

```
// in function modifyArray, "b" points to the original array "a" in memory
void modifyArray( int b[], int sizeOfArray )
{
   // multiply each array element by 2
   for ( int k = 0; k < sizeOfArray; k++ )
      b[ k ] *= 2;
} // end function modifyArray

// in function modifyElement, "e" is a local copy of
// array element a[ 3 ] passed from main
void modifyElement( int e )
{
   // multiply parameter by 2
   std::cout << "Value of element in modifyElement: " << ( e *= 2 )
             << std::endl;
} // end function modifyElement
```

```
// Fig. 6.14: fig06_14.cpp
// Demonstrating the const type qualifier.
#include <iostream>

void tryToModifyArray( const int [] ); // function prototype

int main()
{
   int a[] = { 10, 20, 30 };

   tryToModifyArray( a );
   std::cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
} // end main

// In function tryToModifyArray, "b" cannot be used
// to modify the original array "a" in main.
void tryToModifyArray( const int b[] )
{
   b[ 0 ] /= 2; // compilation error
   b[ 1 ] /= 2; // compilation error
   b[ 2 ] /= 2; // compilation error
} // end function tryToModifyArray
```

Array's Base Address

- Base address: memory location of first array element
- Example: if `num` is one-dimensional array,
  base address of `num` is address (memory location) of `num[0]`
- When passing arrays as parameters, base address of array passed to formal parameter
- Functions cannot return value of type array

Practical – Searching an Array

- Often necessary to determine whether an array contains a **key** value
- Process known as **searching**
- Common methods
  - Linear Search
    - The term "Linear" is used in the text. The standard term that is more commonly used is "Sequential" Search. You may see these terms used synonymously.
  - Binary Search (later course)
- Linear (Sequential) Search
  - Compares each element of an array with the **key**
  - Without additional processing, the array is unsorted (no particular order)
  - Element may be found on first comparison, last comparison or somewhere in between
  - To determine that a value is NOT in an array, all elements must be searched

Practical – Linear (Sequential) Search

```
// Fig. 6.15: fig06_15.cpp
// Linear search of an array.
#include <iostream>

int linearSearch( const int [], int, int ); // prototype

int main()
{
   const int arraySize = 100; // size of array a
   int a[ arraySize ]; // create array a
   int searchKey; // value to locate in array a

   for ( int i = 0; i < arraySize; ++i )
      a[ i ] = 2 * i; // create some data

   std::cout << "Enter integer search key: ";
   std::cin >> searchKey;

   // attempt to locate searchKey in array a
   int element = linearSearch( a, searchKey, arraySize );

   // display results
   if ( element != -1 )
      std::cout << "Found value in element " << element << std::endl;
   else
      std::cout << "Value not found" << std::endl;
} // end main
```

Practical – Linear (Sequential) Search

```
// compare key to every element of array until location is
// found or until end of array is reached; return subscript of
// element if key or -1 if key not found
int linearSearch( const int array[], int key, int sizeOfArray )
{
   for ( int j = 0; j < sizeOfArray; ++j )
      if ( array[ j ] == key ) // if found,
         return j; // return location of key

   return -1; // key not found
} // end function linearSearch
```

- `const` to prevent the function from modifying the array

Practical – Sorting an Array

- Sorting – placing the data into a particular order such as ascending or descending
- Common Sorting Algorithm's
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Selection Sort
- Insertion sort—a simple, but inefficient, sorting algorithm.
- The first iteration of this algorithm takes the second element and, if it's less than the first element, swaps it with the first element (i.e., the program *inserts the second element in front of the first element).*
- The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order.
- At the *i^{th} iteration of this algorithm, the first i elements in the original array will be sorted.*

Practical – Insertion Sort

```
// Fig. 6.16: fig06_16.cpp
// This program sorts an array's values into ascending order.
#include <iostream>
#include <iomanip>

int main()
{
   const int arraySize = 10; // size of array a
   int data[ arraySize ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
   int insert; // temporary variable to hold element to insert

   std::cout << "Unsorted array:\n";

   // output original array
   for ( int i = 0; i < arraySize; ++i )
      std::cout << setw( 4 ) << data[ i ];

   .
   .
   .
//Continued next slide
```

Practical – Insertion Sort

```
// insertion sort
   // loop over the elements of the array
   for ( int next = 1; next < arraySize; ++next )
   {
      insert = data[ next ]; // store the value in the current element

      int moveItem = next; // initialize location to place element

     // search for the location in which to put the current element
      while ( ( moveItem > 0 ) && ( data[ moveItem - 1 ] > insert ) )
      {
         // shift element one slot to the right
         data[ moveItem ] = data[ moveItem - 1 ];
         --moveItem;
      } // end while

      data[ moveItem ] = insert; // place inserted element into the array
   } // end for

   std::cout << "\nSorted array:\n";

   // output sorted array
   for ( int i = 0; i < arraySize; ++i )
      std::cout << setw( 4 ) << data[ i ];

   std::cout << std::endl;
} // end main
```

C-style strings

- Arrays of type `char` are special cases
- C-style string
  - Implemented as an array of type char that ends with a special character, called the “null character”
  - The null character has ASCII value of 0
  - The null character can be written as a literal in code like: ‘`\0`’
  - Every string literal (i.e. something in double quotes) implicitly contains the null character at the end

C-style strings

- Character arrays are used to store C-style strings
- Can initialize a character array with a string literal
  - String literal is a string in double quotes
  - Account for room for the null character when allocating space

```
char name[7] = "Johnny";  //extra space for null character
```

Equivalent to:

```
char name[7] = {'J', 'o', 'h', 'n', 'n', 'y', '\0'};
```

Questions?