ITM 312, Fall 2013

Chapter 7 Lecture Notes

7.1. Arrays Hold Multiple Values

a. Array: variable that can store multiple values of the same type
b. Values are stored in adjacent memory locations
c. Syntax: Declared using `[]` operator: `int tests[5];`
    i. `int` is the data type of the array elements
    ii. Tests is the name of the array
    iii. 5 in `[5]` is the size declarator
    iv. Size declarator (e.g. 5) * number of bytes for the array's datatype = size of array

7.2. Accessing Array Elements

a. Each element in an array is assigned a unique subscript.
b. Subscripts start at **0**
c. The last element's subscript is **n-1** where **n** is the number of elements in the array.
d. Array elements are treated like regular variables
e. Syntax:

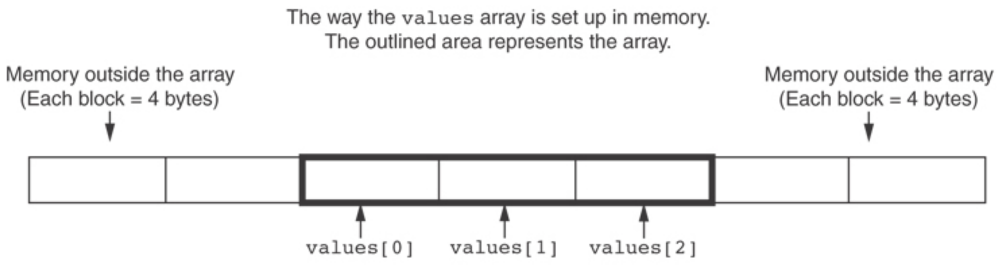
```
cout << tests[0]; // outputs 1st element in array
cin >> tests[1]; //sets 2nd element in array to user input
tests[4] = tests[0] + tests[1]; //sets 5th element in array to
sum of 1st item and 2nd item
```

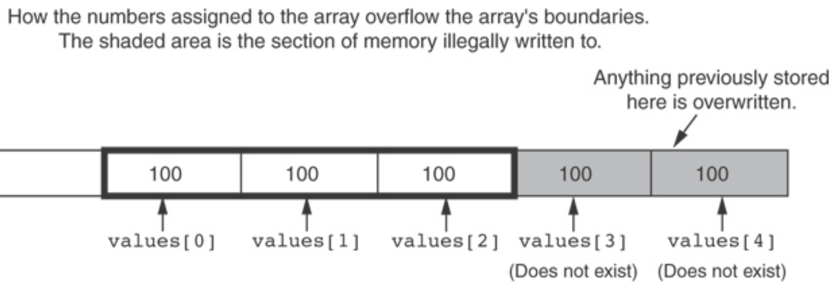
f. Use a `for` loop to get all the items in an array
g. Initialization notes:
    i. Global array: all elements initialized to 0 by default
    ii. Local array: all elements uninitialized* by default (will get error if you try to display an uninitialized array) * - see 7.4d

7.3. No Bounds Checking in C++

a. When you use a value as an array subscript, C++ does not check it to make sure it is a valid subscript.

b. In other words, you can use subscripts that are beyond the bounds of the array. E.g. `int values[5]; int values[6] = 2;` is valid according to the compiler


The way the values array is set up in memory.
The outlined area represents the array.
Memory outside the array (Each block = 4 bytes)
Memory outside the array (Each block = 4 bytes)
values[0]
values[1]
values[2]

c. 
How the numbers assigned to the array overflow the array's boundaries.
The shaded area is the section of memory illegally written to.
Anything previously stored here is overwritten.
100
100
100
100
100
values[0]
values[1]
values[2]
values[3] (Does not exist)
values[4] (Does not exist)

d. Be careful not to use invalid subscripts. Start loops at 0 rather than 1
   i. Doing so can corrupt other memory locations, crash program, or lock up computer, and cause elusive bugs.

7.4. Array Initialization

a. Arrays can be initialized with an initialization list:
```
const int SIZE = 5;
int tests[SIZE] = {79,82,91,77,84};
```
b. The values are stored in the array in the order in which they appear in the list.

c. The initialization list cannot exceed the array size.

d. If array is initialized with fewer initial values than the size declarator, the remaining elements will be set to 0 (even if it's a local variable)

e. Can determine array size by the size of the initialization list (leave `[]` blank)

f. Must use either array size declarator or initialization list at array definition

7.5. Processing Array Contents

a. Array elements can be treated as ordinary variables of the same type as the array

b. When using `++`, `--` operators, don't confuse the element with the subscript:

   `tests[i]++; // add 1 to tests[i]`

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

c. To copy one array to another, use a loop to go element-by-element

d. You can display the contents of a character array by sending its name to cout:

   `char fName[] = "Henry";`

   `cout << fName << endl; //But, this ONLY works with character arrays!`

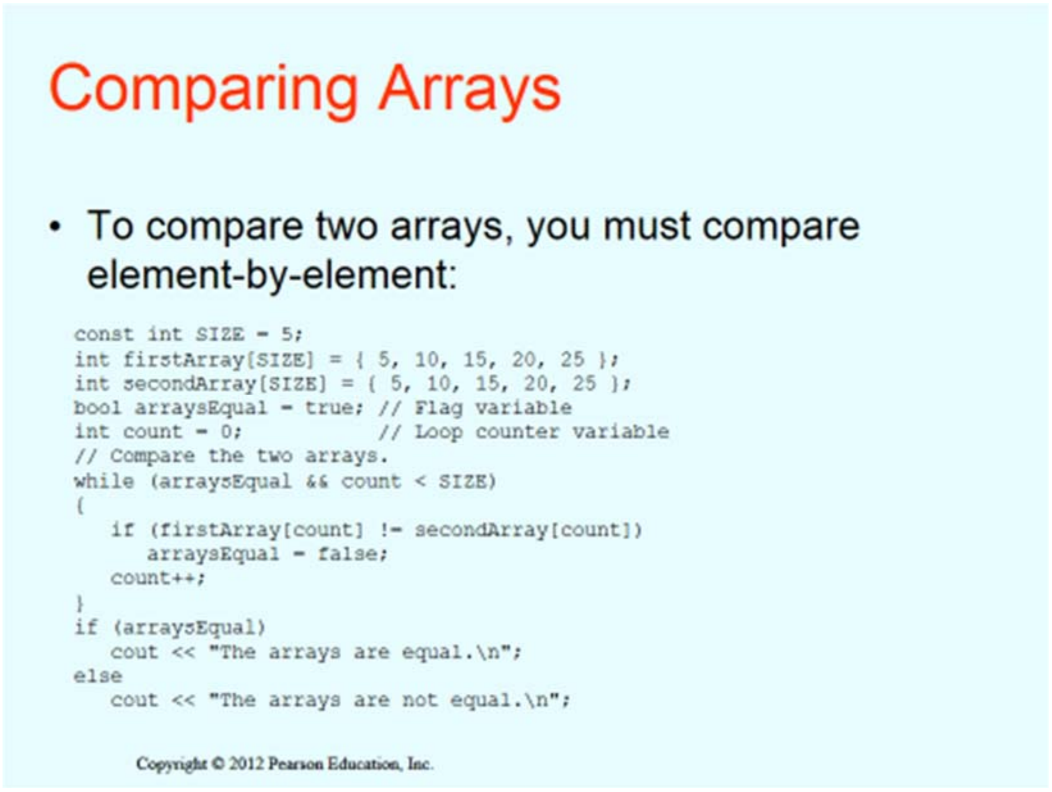
e. For all other array types, print element-by-element (w/ a `for` loop)

f. Use loops to find sum, min, max of array

g. If it is unknown how much data an array will be holding:

   i. Make the array large enough to hold the largest expected number of elements.

   ii. Use a counter variable to keep track of the number of items stored in the array.

h. 

Comparing Arrays

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

```
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";
```

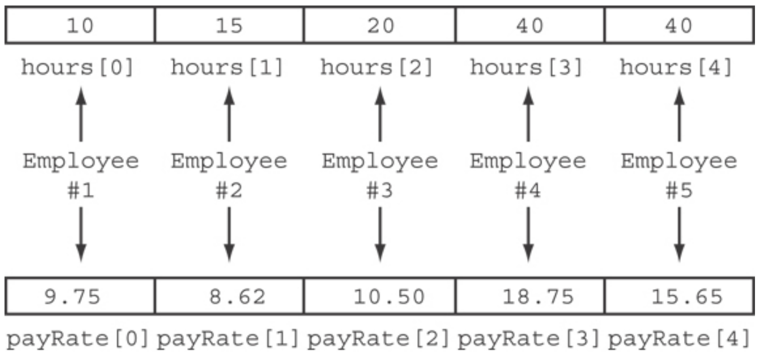
7.6. Using Parallel Arrays

a. Parallel arrays: two or more arrays that contain related data

b. A subscript is used to relate arrays: elements at same subscript are related

c. Arrays may be of different types

d. See program 7-12 for an example. The `hours` and `payRate` arrays are related through their subscripts:



7.7. Arrays as Function Arguments

a. To pass an array to a function, just use the array name:
```
showScores(tests);
```

b. To define a function that takes an array parameter, use empty `[]` for array argument:
```
void showScores(int []); // function prototype
void showScores(int tests[]) // function header
```

  i. We don't define a size in the function prototype+header because we want to accept all array sizes and not limit ourselves

c. 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);
```

  i. Array size must also be reflected in prototype, header:
```
void showScores(int [], int); // function prototype
void showScores(int tests[], int size) // function header
```

d. Modifying arrays in functions

  i. Array names in functions are like reference variables – changes made to array in a function are reflected in actual array in calling function

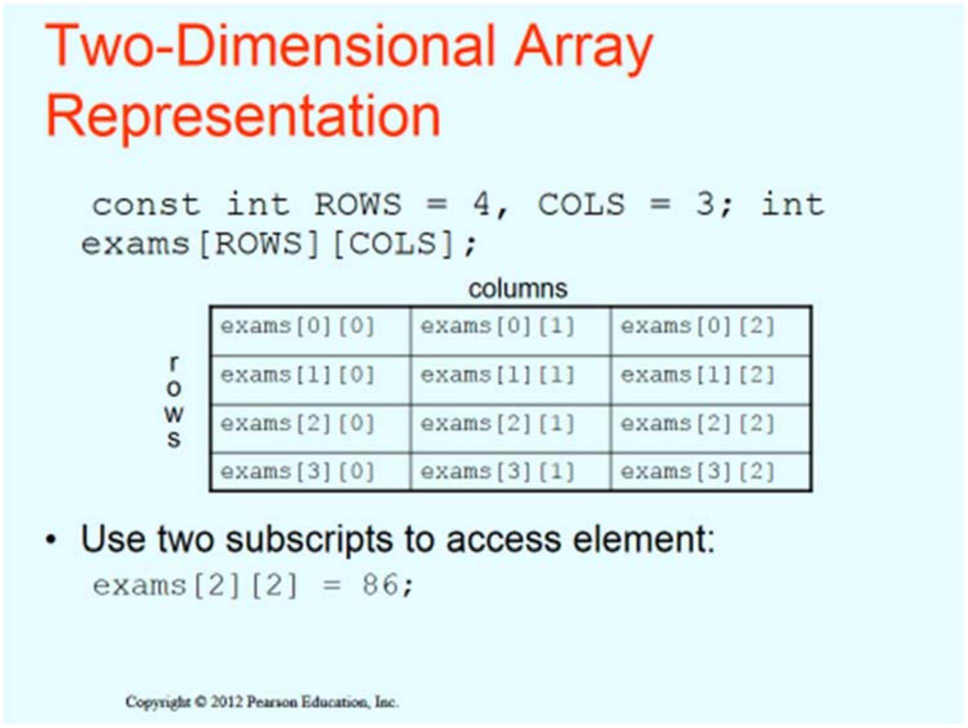
  ii. Need to exercise caution that array is not inadvertently changed by a function

7.8. Two-Dimensional Arrays

a. Can define one array for multiple sets of data
b. Like a table in a spreadsheet
c. Use two size declarators in definition:

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

d. First declarator is number of rows; second is number of columns

e. 

Two-Dimensional Array Representation

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

rows \ columns			
	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 element:

```
exams[2][2] = 86;
```

f. You need n loops to access all the elements of the n-dimensional array
g. Two-dimensional arrays are initialized row-by-row:

```
const int ROWS = 2, COLS = 2;
int exams[ROWS][COLS] = {
                          {col1_1, col2_1},
                          {col1_2, col2_2}
                        };
```

col1_1	col2_1
col1_2	col2_2

h. Can omit inner { }, some initial values in a row – array elements without initial values will be set to 0 or NULL
i. Refer to slides for how to process 2D arrays

7.9. Arrays with Three or More Dimensions – did not cover
7.10. Introduction to the STL vector – did not cover