# CISC181 Introduction to Computer Science

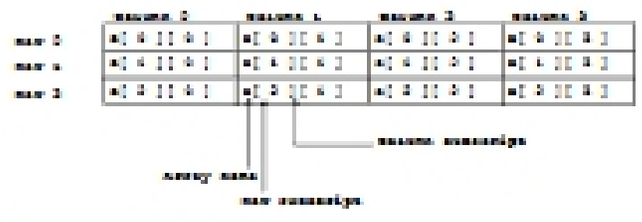
Dr. McCoy

Lecture 12
October 8, 2009



## 4.9 Multiple-Subscripted Arrays

- Multiple subscripts
  - `a[ i ][ j ]`
  - Tables with rows and columns
  - Specify row, then column
  - "Array of arrays"
    - `a[0]` is an array of 4 elements
    - `a[0][0]` is the first element of that array





## 4.9 Multiple-Subscripted Arrays

- To initialize
  - Default of 0
  - Initializers grouped by row in braces

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

| | |
|---|---|
| 1 | 2 |
| 3 | 4 |

How 0 How 1

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

| | |
|---|---|
| 1 | 0 |
| 3 | 4 |



## 4.9 Multiple-Subscripted Arrays

- Referenced like normal

```
cout << b[ 0 ][ 1 ];
```

| | |
|---|---|
| 1 | 0 |
| 3 | 4 |

  - Outputs 0
  - Cannot reference using commas

```
cout << b[ 0, 1 ];
```

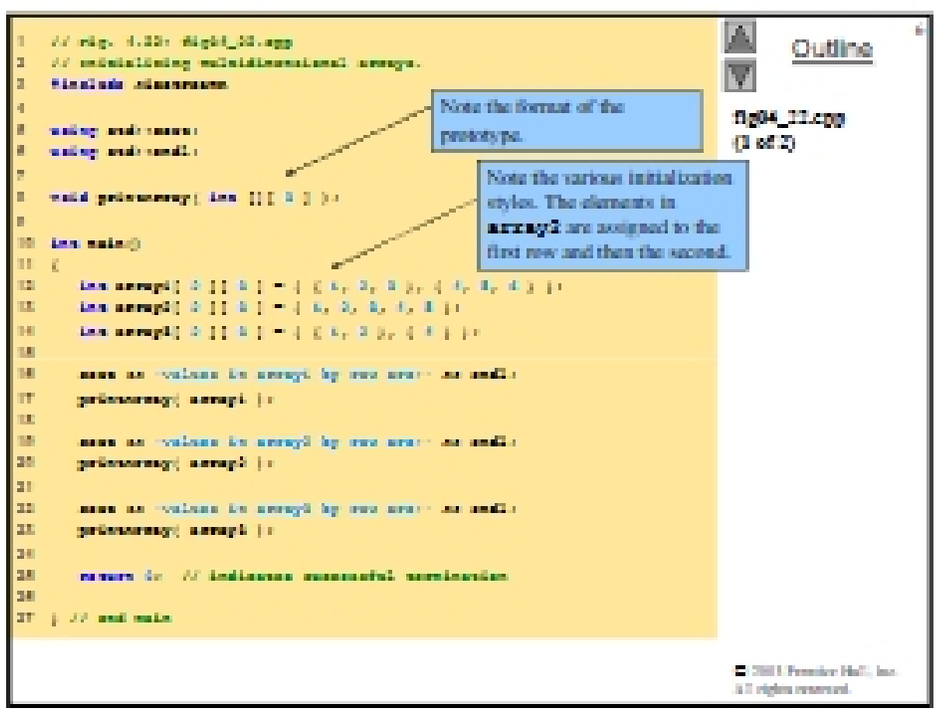
    - Syntax error
- Function prototypes
  - Must specify sizes of subscripts
    - First subscript not necessary, as with single-scripted arrays

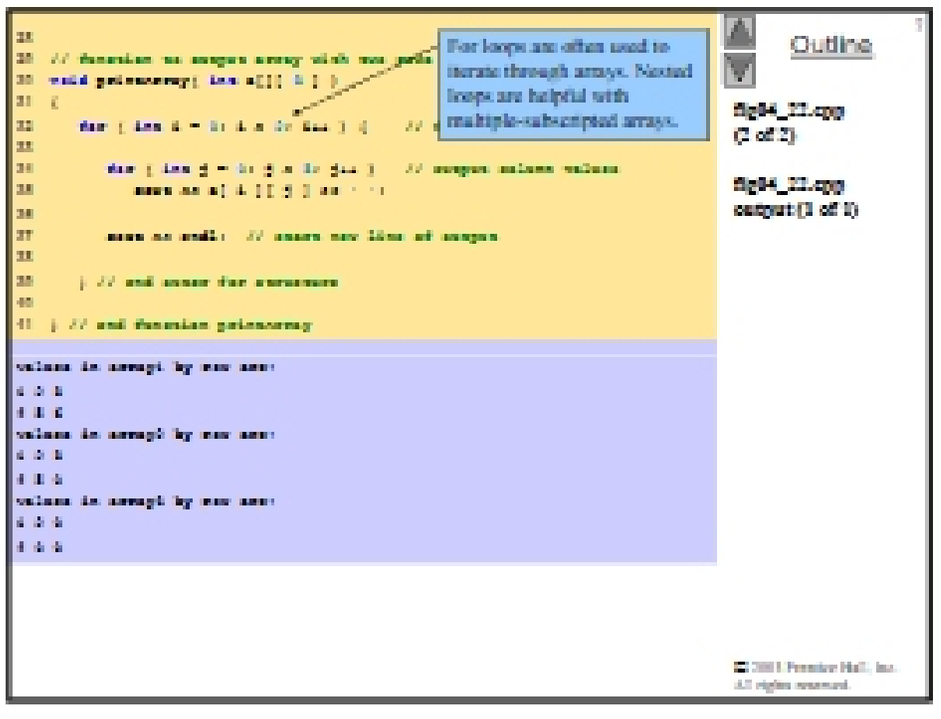
```
void printArray( int [][ 3 ] );
```

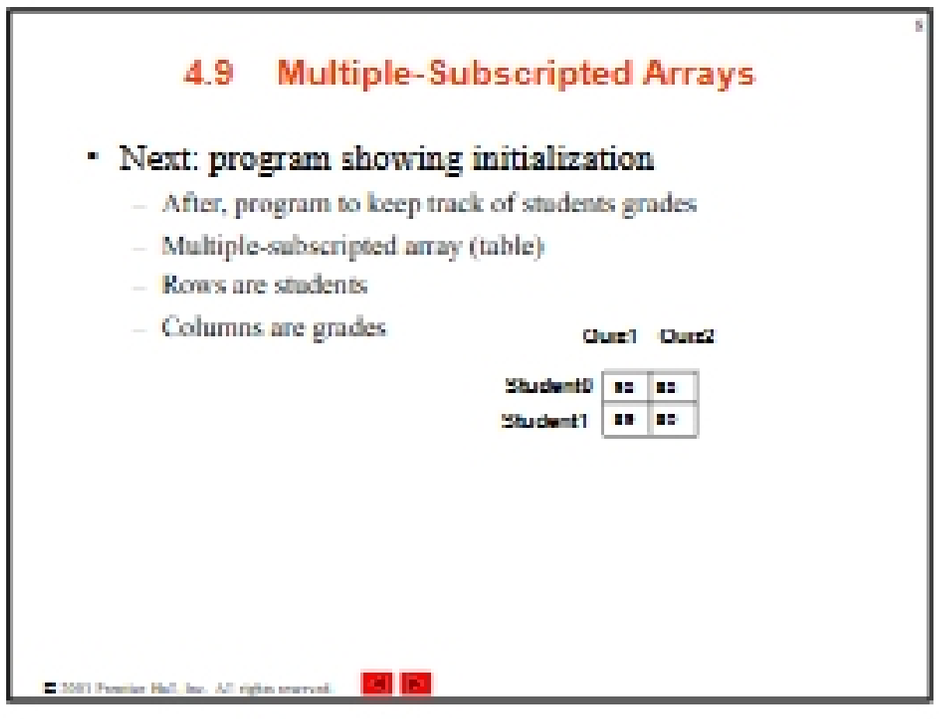


## NOTE PROBLEM WITH NEXT FUNCTION

- Normally when an array is passed as an argument to a function, you also pass the number of elements in that array.
- When an array with multiple subscripts is passed to a function – it is passed as an array (giving no "number" of elements) of arrays (whose size it given).
- But, in the following function the max index is left off (the function only works for one size array.

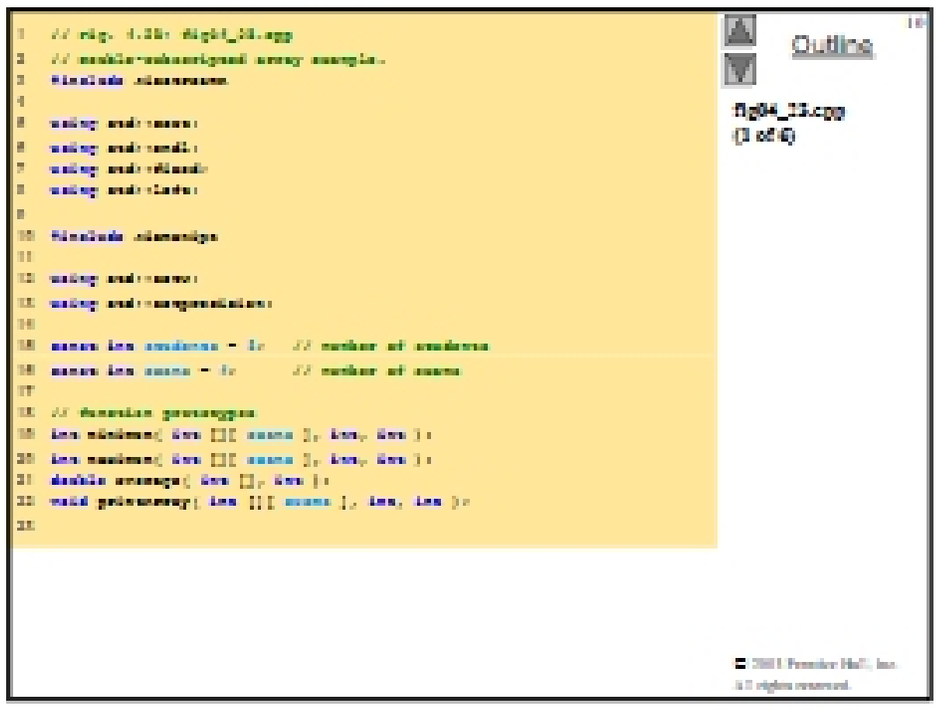


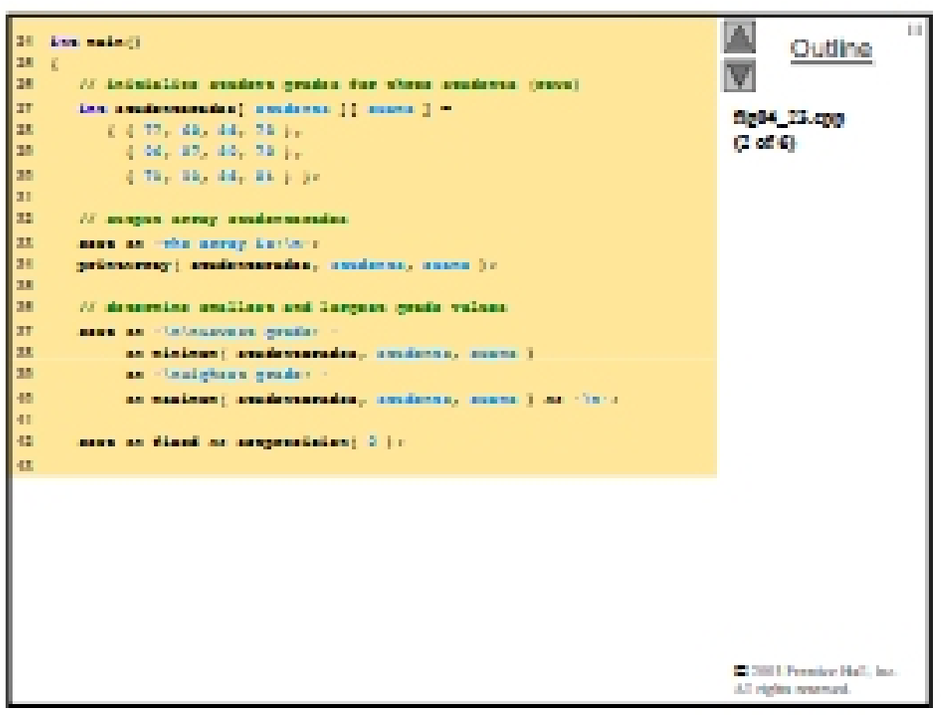


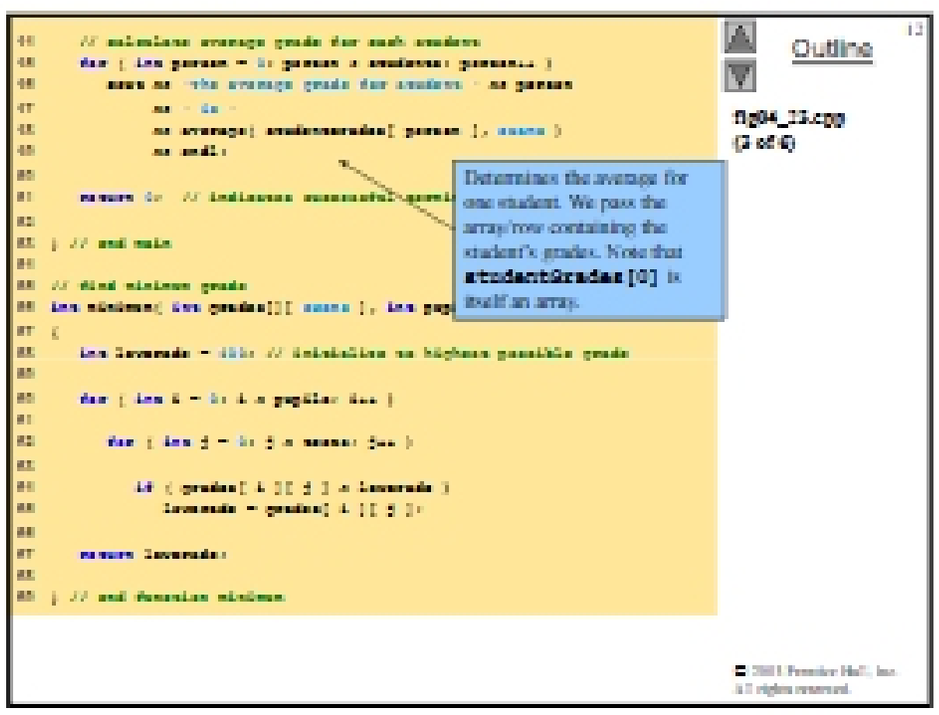




## NOTE POOR PROGRAMMING STYLE

- Lack of documentation with function prototypes.
- Lack of variable names in function prototypes.
- NOTE: Can't at all tell what function does by looking at the prototype. That is WRONG.



```
55
56 // find maximum grade
57 int maximum( int grades[][ exams ], int pupils, int exams )
58 {
59    int highGrade = 0;  // initialize to lowest possible grade
60
61    for ( int i = 0; i < pupils; i++ )
62
63       for ( int j = 0; j < exams; j++ )
64
65          if ( grades[ i ][ j ] > highGrade )
66             highGrade = grades[ i ][ j ];
67
68    return highGrade;
69
70 } // end function maximum
71
```



Outline

fig04_23.cpp
(4 of 6)



```
71 // determine average grade for particular student
72 double average( int studentGrades[], int exams )
73 {
74    int total = 0;
75
76    // total all grades for one student
77    for ( int i = 0; i < exams; i++ )
78       total += studentGrades[ i ];
79
80    return static_cast< double >( total ) / exams;  // average
81
82 } // end function maximum
```



Outline

fig04_23.cpp
(5 of 6)



```
85
86 // print the array
87 void printArray( int grades[][ exams ], int pupils, int exams )
88 {
89    // set left justification and output column heads
90    cout << left << "                 [0]  [1]  [2]  [3]";
91
92    // output grades in tabular format
93    for ( int i = 0; i < pupils; i++ ) {
94
95       // output label for row
96       cout << "\nstudentGrades[" << i << "] ";
97
98       // output one grades for one student
99       for ( int j = 0; j < exams; j++ )
100         cout << setw( 5 ) << grades[ i ][ j ];
101
102   } // end outer for
103
104 } // end function printArray
```



Outline

fig04_23.cpp
(6 of 6)



```
The array is:
                 [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```



Outline

fig04_23.cpp
output (1 of 1)



## 4.8 Searching Arrays: Linear Search and Binary Search



- Search array for a key value
- Linear search
  - Compare each element of array with key value
    - Start at one end, go to other
  - Useful for small and unsorted arrays
    - Inefficient
    - If search key not present, examines every element



## 4.8 Searching Arrays: Linear Search and Binary Search



- Binary search
  - Only used with sorted arrays
  - Compare middle element with key
    - If equal, match found
    - If key < middle
      - Repeat search on first half of array
    - If key > middle
      - Repeat search on last half
  - Very fast
    - At most N steps, where $2^N$ > # of elements
    - 30 element array takes at most 5 steps
      $2^5 > 30$