CIS 22A – Lecture 14

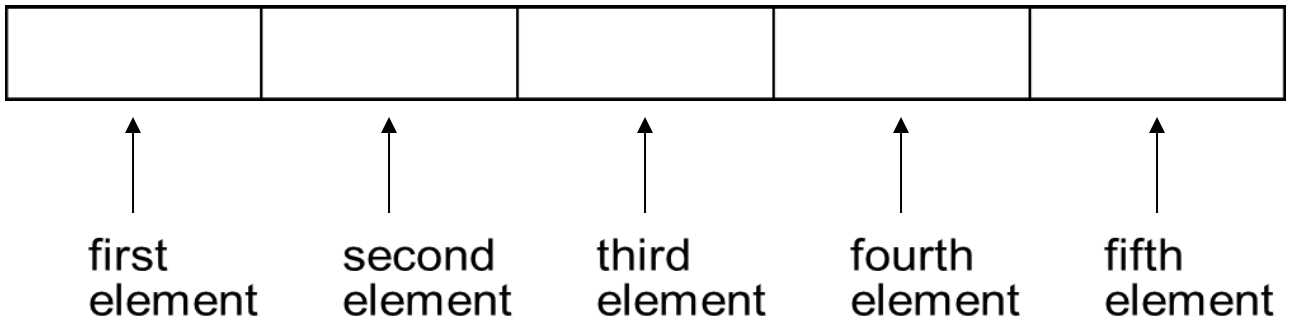
Manish Goel

Arrays

1. Arrays – Variables to hold multiple values of same data-type
2. Can be likened to “collections or groups of related data”
3. Defined as:

```
int myInts [5];
double myDoub [6];
char myChars [9];
string myStr [SIZE]; // where const int SIZE = 3;
```

4. The declaration has a data-type, a name and a “constant” size that defines how many values can be stored in array.
5. The declaration allocates memory for array as



Arrays (cont.)

1. Size of an array = sizeof(data-type) * number of elements
2. So size of following:

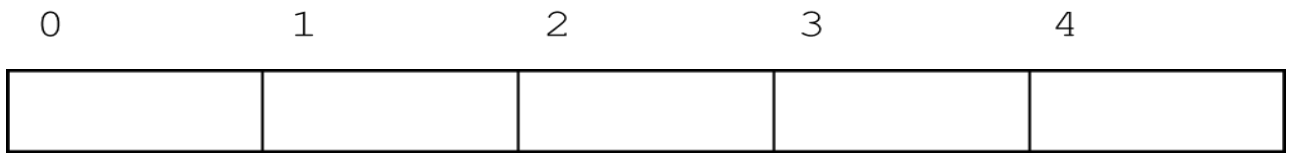
```
int myInts [5];        4 byts * 5 = 20 bytes
double myDoub [6];     4 byte * 6 = 24 bytes
char myChars [9];      1 byte * 9 = 9 bytes

string myStr [SIZE]; // where const int SIZE = 3;
sizeof string array is the sum of all string lengths
```

3. The size declarator can only be either a positive integer or a `const int` named variable, so that the program knows how much memory to allocate
4. Use of named constants helps to make program changes easier – don't need to change multiple lines where the size is referred

Array Elements

1. Elements of an array store the literals of the array's data-type and can be used individually
2. Elements have subscripts or index so they can be used
3. Subscripts start at `0`
4. Subscripts end at `n-1` where `n` is the size declarator or the number of elements in array

```
int myInts [5];
```



IMPORTANT – this causes a one-off relationship because array subscripts don't start from 1 to n

Array Elements (cont.)

1. Arrays are accessed via individual elements
2. Array elements can be used as variables in expressions or for input / output from screen or files

```
const int ARR_SIZE = 5;     // declare array size
int myInts [ARR_SIZE];      // declare array

for (int i = 0; i < ARR_SIZE; i++)
        cin >> myInts[i];   // assign array elements
                            // from screen input
int iSum = myInts[1] + myInts[3]; // sum 2 elements

for (int i = 0; i < ARR_SIZE; i++)
        cout << myInts[i] << endl; // print to screen

cout << iSum;               // print the sum
```

Array Elements – Things to remember

1. Always loop through array elements

```
cout << myInts;      // not legal
```

2. Declaring – always with positive integer size
3. Accessing – either a positive number or a variable subscript

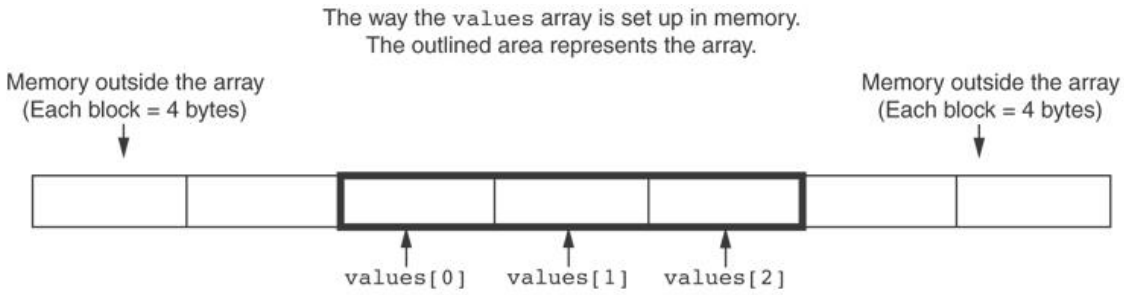
4. For number type arrays,
   - Global array: all elements are initialized to 0 by default
   - Local array: NO elements are initialized and needed to be done

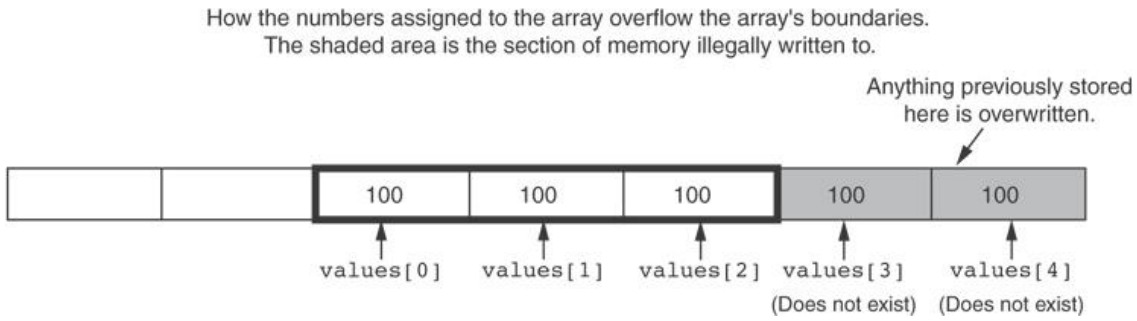
5. No bounds checking in C++, i.e. invalid subscripts can be used to reach beyond array boundary;
6. DANGER – memory of other variables or even other programs can be corrupted by using invalid subscripts

No Bounds Checking

- Program for 7-5

```
 9     const int SIZE = 3;  // Constant for the array size
10     int values[SIZE];    // An array of 3 integers
11     int count;           // Loop counter variable
12
13     // Attempt to store five numbers in the three-element array.
14     cout << "I will store 5 numbers in a 3 element array!\n";
15     for (count = 0; count < 5; count++)
16        values[count] = 100;
```





Array Initialization

1. Arrays can be initialized by accessing and assigning each element in a loop as shown earlier.
2. Arrays can also be initialized using an initialization list

```
const int ARR_SIZE = 5;
int myInts [5] = {11, 16, 23, 44, 98};
```

3. Values are stored in order of list and list cannot exceed the number of elements
4. Initialization list can be less than the number of elements – all other elements set to ‘0’.

```
int myInts [5] = {11, 16, 23};
```

5. If array size not declared, then initialization list sets size of array implicitly

```
int myInts [] = {11, 16, 23, 44, 98};
```

Some Array Operations

1. Sum and Average of array

```
int tnum;
double average, sum = 0;
for(tnum = 0; tnum < SIZE; tnum++)
       sum += tests[tnum];
average = sum / SIZE;
```

2. Finding the highest and lowest element in the array

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

Processing Array Contents

1. Array elements are ordinary variables of same data-type as the array itself
2. Printing arrays – all arrays have to be printed element by element

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

3. Only exception – `char` arrays which can be printed by name

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

4. Be careful when using ++, -- operators – they are allowed on both the element value as well as subscript

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

Array – Copy or Compare

1. Array Assignment or Copy Array from one to another – needs to be done element by element

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

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

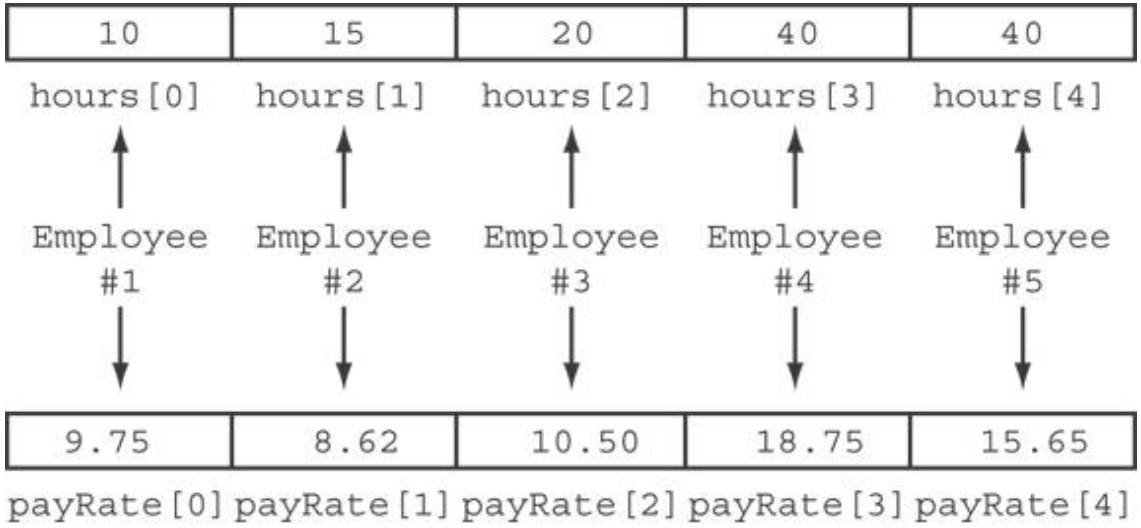
2. Comparing arrays – compare each element – arrays are equal only if all elements and size of array match

```
while (arraysEqual && count < SIZE)
{
   if (firstArray[count] != secondArray[count])
      arraysEqual = false;
   count++;
}
```

Parallel Arrays – simplistic database?

1. Parallel Arrays are two or more arrays (can be of different data-types) that contain related data
2. Same subscript used to traverse all arrays simultaneously
   - e.g. Student_ID, testScore and testGrade, or,
   - Employee_ID, payRate and hours

The `hours` and `payRate` arrays are related through their subscripts:



Two-dimensional arrays

1. Like a table in a spreadsheet
2. Two size declarators – first declarator is number of rows, second is number of columns

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

3. Use two subscripts to access elements

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

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]

Two-dimensional arrays (cont.)

1. 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

2. Can omit inner `{ }`, some initial values in a row – array elements without initial values will be set to `0` or `NULL`

3. To process, array elements – use nested loops – outer for rows, inner for columns (vice-versa)

```
for (int row = 0; row < NUM_ROWS; row++)
{
   for (int col = 0; col < NUM_COLS; col++)
      cout << numbers[row][col];
}
```

Multi-dimensional arrays

1. Can define arrays with many dimensions

```
short rectSolid[2][3][5];
double timeGrid[3][4][3][4];
```

2. To access array elements: # of dimensions = # of loops

3. To visualize,
   - One dimensional array ←→ Points on a line
   - Two dimensional arrays ←→ Points on a graph
   - Three dimensional arrays ←→ Points in space
   - Four dimensional arrays ←→ Points in space along a timeline