ENGINEERING PROGRAMMING I

Chapter 08-Part 2

2017

The `typedef` Statement

- Creates an alias for a simple or structured data type
- Format:

```
typedef existingType newName;
```

- Example:

```
typedef unsigned int Uint;
Uint tests[ISIZE]; // array of
                   // unsigned ints
```

Uses of `typedef`

- Used to make code more readable
- Can be used to create alias for array of a particular type

```
// Define yearArray as a data type
// that is an array of 12 ints
typedef int yearArray[MONTHS];

// Create two of these arrays
yearArray highTemps, lowTemps;
```

Arrays as Function Arguments

- To define a function that has an array parameter, use empty `[]` to indicate the array argument
- To pass an array to a function, just use the array name

```
// Function prototype
void showScores(int []);

// Function header
void showScores(int tests[])

// Function call
showScores(tests);
```

Passing an Array Element

- Passing a single array element to a function is no different than passing a regular variable of that data type
- Function does not need to know that the value it receives is coming from an array

```
displayValue(score[i]);     // call
void displayValue(int item) // header
{  cout << item << endl;
}
```

Passing an Entire Array

- Use the array name, without any brackets, as the argument
- Can also pass the array size so the function knows how many elements to process

```
showScores(tests, 5);          // call
void showScores(int[], int); // prototype
void showScores(int A[],
                int size) // header
```

Using `typedef` with a Passed Array

Can use **`typedef`** to simplify function prototype and heading

```
// Make intArray an integer array
// of unspecified size
typedef int intArray[];

// Function prototype
void showScores(intArray, int);

// Function header
void showScores(intArray tests,
                int size)
```

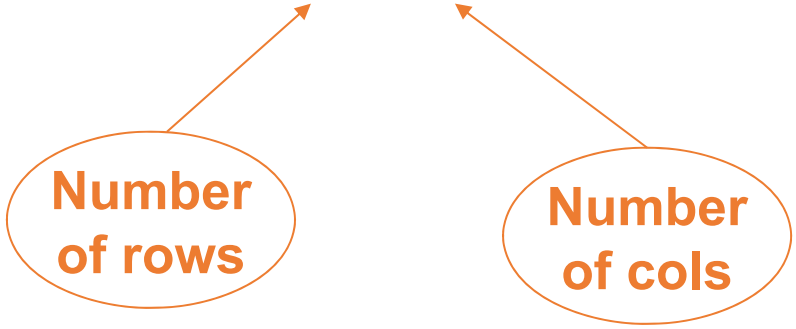
Modifying Arrays in Functions

- Array parameters in functions are similar to reference variables
- Changes made to array in a function are made to the actual array in the calling function
- Must be careful that an array is not
- inadvertently changed by a function

Two-Dimensional Arrays

- Can define one array for multiple sets of data
- Like a table in a spreadsheet
- Use two size declarators in definition

```
int exams[4][3];
```



Two-Dimensional Array Representation

```
int exams[4][3];
```

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;
```

Initialization at Definition

- Two-dimensional arrays are initialized row-by-row

```
int exams[2][2] = { {84, 78},
                    {92, 97} };
```

84	78
92	97

- Can omit inner **{ }**

Passing a Two-Dimensional Array to a Function

- Use array name as argument in function call

```
getExams(exams, 2);
```

- Use empty `[]` for row and a size declarator for col in the prototype and header

```
// Prototype, where NUM_COLS is 2
void getExams(int[][NUM_COLS], int);

// Header
void getExams
        (int exams[][NUM_COLS], int rows)
```

Using `typedef` with a Two-Dimensional Array

Can use `typedef` for simpler notation

```
typedef int intExams[][2];
   ...
// Function prototype
void getExams(intExams, int);

// Function header
void getExams(intExams exams, int rows)
```

2D Array Traversal

- Use nested loops, one for row and one for column, to visit each array element.
- Accumulators can be used to sum the elements row-by-row, column-by-column, or over the entire array.

Arrays with Three or More Dimensions

- Can define arrays with any number of dimensions

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

- When used as parameter, specify size of all but 1st dimension

```
void getRectSolid(short [][3][5]);
```