Multiple dimension arrays and passing arrays as function outputs

CS 10A – ADVANCED ARRAY CONCEPTS

Allen Zhao – SRJC CS 10A

Multi-Dimensional Arrays

- Arrays can have multiple dimensions. That is, each array element can be an array itself. This can continue infinitely, inception-style, depending on what your system will allow.
- Generally, use of 2D and 3D arrays are fairly common, but it's not recommended to use anything above 3D, unless you're doing advanced modern physics. The number of elements in a multi-dimensional array grows exponentially.
- At some point, using arrays with too many dimensions will slow down your computer by a significant margin due to limitations of memory access time.

Declaring Multi-Dimensional Arrays

```
int main()
{
        int size_row = 5, size_column = 2;
        double arr_2d_0[size_row][size_column];        // No value initialization

        // a 2D array is easily visualized as a grid
        // for more dimensions, just add another [ ] to the declaration line
        // The same declaration rules and methods of 1D arrays (generally) apply here

        double arr_2d_1[][2] = {{2, 3}, {1, 1}, {5, 0}, {4, 2}};        // Specify one dimension only
        // In any instance where a multi-dimensional array is used, only the first level can be arbitrary
        double arr_2d_2[][2] = {2, 3, 1, 1, 5, 0, 4, 2};        // The compiler can auto-group your list

        return 0;
}
```

Declaring Multi-Dimensional Arrays – Method 3

```
int main()
{
        int ** arr_2d = NULL;        // Double asterisks
        int size_row, size_column;

        cout << “Enter Dimensions (rows, cols): ”;
        cin >> size_row >> size_column;

        arr_2d = new int * [size_row];        // Sets the number of 1D arrays for the 2D array
        for(int i = 0; i < size_row; i++)
                arr_2d[i] = new int[size_column];   // Declare a new array for every index

        return 0;
}
```

Accessing Multi-Dimensional Arrays

```
int main()
{
        int size_row = 4, size_column = 3;
        double arr_2d_0[size_row][size_column];

        // Accessing an individual element
        double single_value = arr_2d_0[1][2];

        // Going through all elements sequentially
        for(int i = 0; i < size_row; i++)
                for(int j = 0; j < size_column; j++)
                        cout << arr_2d_1[i][j] << endl;
        // Further nest more loops for additional dimensions if necessary
        return 0;
}
```

Size of Multi-Dimensional Arrays

Program

```
int main()
{
        double arr_2d[4][2];
        // sizeof() works the same as before
        // still does not work on arrays declared by Method 3

        cout << sizeof(arr_2d) << endl;
        cout << sizeof(arr_2d[0]) << endl;
        cout << sizeof(arr_2d[0][0]) << endl << endl;
        cout << sizeof(arr_2d)/sizeof(arr_2d[0]) << endl;
        cout << sizeof(arr_2d[0])/sizeof(arr_2d[0][0]) << endl;
        cout << sizeof(arr_2d)/sizeof(arr_2d[0][0]) << endl;

        return 0;
}
```

Console

```
➢ ./a.exe
64
16
8

4
2
8
```

Applications

- As you can imagine, 2D arrays are excellent for holding coordinates. Coordinates of any dimension (x, y, z, t) are easily represented in 2D arrays.
  - Do not confuse this with 3D+ arrays! 2D arrays can easily represent coordinates in either 2D, 3D, or 4D.
- Another common application for 2D arrays is representing color. In most computer systems, a single color is represented by 3 separate hex values.
- 3D arrays can generally be used to group 2D arrays into separate categories if necessary.
- If you value your sanity, avoid using arrays 4D and above.

Pointers - A Short Introduction

- Every variable in your program is reserved memory space on your computer that holds the value you want to use.
- To know where the value is stored when it’s time to access it, every variable is assigned a memory address, which is usually just some hex value.
- These memory addresses can be randomly assigned for standalone variables, but all elements within an array have addresses immediately next to each other, and can be incremented through.
- We can store these addresses in a what’s called a pointer.
- Asterisks are used to mark pointer type variables.

Returning Arrays as Function Outputs

- Arrays can’t pass around their values like standalone variables between functions. They actually are passed around via pointers. This is why, once we pass an array into a function, we can’t determine their size using sizeof().
- To pass an array back as a function output, we need to pass it as a pointer of the same type. (Strings are the exception.) Unfortunately, because functions can only have one output, we can’t pass the array size back with it.
  - As a result, our use of this method is limited to arrays whose size we already know, i.e. an array we originally passed in as a parameter.
- Multi-dimensional arrays are unsupported in this regard.
  - To get around this, simply use a loop to put each inner array element through the function one by one.

Function – Array Output

```
int * func_arrOut(int arr[], int size)          // This function returns an integer pointer, which serves as an integer array for us
{
        int * sub_arr = NULL;          // Declare a new array via Method 3
        sub_arr = new int[size];
        for(int i = 0; i < size; i++)
                sub_arr[i] = arr[i] + 1;          // Increment all values in the given array by 1
        return sub_arr;                    // Since input arrays are pointers, you can write in changes to arr[] directly, but that modifies the original data
}

int main()
{
        int length = 6;
        int ex[length] = {1, 2, 3, 4, 5, 6};
        int * p;          // Declare the pointer to hold 1D arrays of unknown size

        p = func_arrOut(ex, length);
        for(int i = 0; i < length; i++) // Since returned arrays will not have a defined size, you'll have to hold the size of the array given
                cout << p[i] << endl;          // Accessing values from pointer works the same as regular arrays
        // Output will be 2, 3, 4, 5, 6, 7 for p[], and ex[] still holds 1-6
        return 0;
}
```