Unit 19: Multi-Dimensional Arrays

Learning Objectives

After completing this unit, students should:

- be comfortable reading and writing code that involves multi-dimensional arrays
- be able to declare and allocate multi-dimensional arrays on the stack and the heap
- understand how we can pass a multi-dimensional array into a function
- understand the differences in C syntax, between an array of pointers and a pointer to an array.
- be able to declare and allocate jagged arrays.

Fixed Length 2D Array

At the beginning of Unit 15, we say that an array can hold one or more values of some type T. T can also be an array. So, we can have an array of array of `long`, for instance.

```
1  long matrix[10][20];
```

Here, we have an array of 10 elements, and each element is an array of 20 `long` values.

MATRIX

When we access the elements in the array, we can use the notation `matrix[i][j]` (which is actually `(matrix[i])[j]`).

Such an array is called a two-dimensional array, or 2D array. We can have a 3D array, 4D array, and so on.

how to declare

We have seen three types of arrays. On the stack, we have a fixed-length array and variable-length array. On the heap, we have dynamically allocated arrays. A 2D array can mix different types of array. Since we discourage the use of variable length array, we will focus on fixed-length arrays and dynamically allocated arrays only.

The example `matrix` above is a fixed-length array. In the memory, a continuous space of 200 `long` values have been allocated, and we can visualize this is having 10 rows of `long` array, each array contains 20 columns of `long` values.

when calling matrix == &matrix[0][0]

With array decay, when we use the first level index of the array, `matrix[i]`, this is equivalent to `&matrix[i][0]` (the address of the first element in matrix[i]), and it has the type `long *`. Thus, if we want to pass individulal row of a 2D array into a function, we have two options for parameters declaration:

- `void bar(long num_of_cols, long* matrix_row) { .. }`
- `void bar(long num_of_cols, long matrix_row[]) { .. }`

This is similar to a 1D array since `matrix[i]` is a 1D array. ?????

We can then invoke the function `bar` like:

```
1  bar(20, matrix[i]);
```

Things get a bit tricky when we want to pass a 2D array into a function. By array decay, when we use the array name `matrix`, it is equivalent to `&matrix[0]`, which is the address of the first element in `matrix`, which is the address of an array of 20 `long` elements.

We can declare the parameter for a function using either one of the following:

- `void qux(long num_of_rows, long num_of_cols, long (* matrix_row)[20]) { .. }`
- `void qux(long num_of_rows, long num_of_cols, long matrix_row[][20]) { .. }`

and call it as:

```
1  qux(10, 20, matrix);
```

intended

Note the parenthesis in the declaration of `long (* matrix_row)[20]`. This declaration is different from `long* matrix_row[20]` (which is actually `long *(matrix_row[20])` Let's decipher this:

big difference

- `long *(matrix_row[20])` actually means `matrix_row` is an array of 20 pointers to `long` values. referencing to the 1st element of the 20 pointers,
  - but to note that matrix_row is also in an array, therefore 2d array
- `long (* matrix_row)[20]` actually means `matrix_row` is a pointer to an array of 20 `long` values.

It is also important to note that, we cannot omit the number `20` in the `[]` since it is part of the type information. A pointer to an array of 20 `long` values is treated as a different type than a pointer to an array of 19 `long` values in C.

Fixed Size Array of Dynamically Allocated Array

Suppose that we know only one of the dimensions of the array in advance, but not the other dimension. We can allocate a fixed-length array for the known dimension, and allocate the other dimension dynamically using `calloc`. We can declare an array like this:

here is a fixed row array (in the stack)
but with variably declared columns (in the heap)

```
double *buckets[10];
long num_of_cols = cs1010_read_long();
for (long i = 0; i < 10; i += 1) {
  buckets[i] = calloc(num_of_cols, sizeof(double));
}
```

Here, `bucket` is a 1D array of 10 pointers to `long`. So, we can easily pass `bucket` to a function just like any other 1D array:

double pointers????

- `void baz(long num_of_rows, long num_of_cols, long **bucket)`
- `void baz(long num_of_rows, long num_of_cols, long *bucket[])`

array of double pointers??

Access such individual elements in such type of array is no different from accessing a fixed-length 2D array: we use `bucket[i][j]`.

*buckets = gives the pointer of the double array
**buckets = gives the value of the double array

Remember to free the allocated memory after we are done:

```
for (long i = 0; i < 10; i += 1) {
  free(buckets[i]);
}
free(buckets);
```

Dynamically Size 2D Array

Suppose that we do not know both dimensions in advance, then we can allocate both dimensions of the array dynamically on the heap.

creating both on the heap, to use later

creating # rows

creating columns in the row

```
double **canvas;
long num_of_rows = cs1010_read_long();
long num_of_cols = cs1010_read_long();
canvas = calloc(num_of_rows, sizeof(double *));
for (long i = 0; i < num_of_rows; i += 1) {
  canvas[i] = calloc(num_of_cols, sizeof(double));
}
```

Passing such an array into a function is no different from a 2D array where only one dimension is dynamically allocated above.

Remember to free the allocated memory for both dimensions after we are done:

```
for (long i = 0; long i < num_of_rows; i += 1) {
  free(canvas[i]);
```

```
3  }
4  free(canvas);
```

Note that during the deallocation of memory, we need to do it in the reverse order of memory allocation. If we call `free(canvas)` first, we are no longer guaranteed to be able to access `canvas[i]` with the correct pointers inside, so calling `free(canvas[i])` after `free(canvas)` might lead to an error.

Jagged Array

One advantage of using a dynamically allocated array is that it allows a jagged 2D array, where each row has a different size. The example below allocate memory for a 2D array that is shaped like a half-square: the first row has one element, the second row two elements, the third row three, and so on.

note

```
1  double *half_square[10];
2  for (long i = 0; i < 10; i += 1) {
3    half_square[i] = calloc(i+1, sizeof(double));
4  }
```

row 1 = 1 element
row 2 = 2 element
row 3 = 3 element

Initializing a Multidimensional Array

Just like a 1D-array, we can initialize a multi-dimensional array with initializers during declaration:

see during lesons

```
1  long matrix[3][3] = {
2      {1, 0, -1},
3      {-1, 1, 0},
4      {0, -1, 1}
5  };
```

Note that we use nested `{` and `}` here. There are other variations to the syntax above, which you may read up on your own if you are interested as we do not need to write complex initializers for multi-dimensional arrays that often.

Problem Set

Problem 19.1

Write two functions described below. Show how you would declare the parameters to each function and how you would call each function.

a) Write a function `add` that performs 3x3 matrix addition. The function should operate on 3x3 matrices of `long`, takes in three parameters, the first two are the operands for addition and the third is the result.

b) Write a function `multiply` that performs 3x3 matrix multiplication. The function should operate on 3x3 matrices of `long`, takes in three parameters, the first two are the operands for multiplication and the third is the result.

Problem 19.2

We need to represent the distance in km between every major city in the world. Let's label every city with a number, ranging from 0 .. $n-1$, where n is the number of cities. The distance between city i and j is the same as the distance between city j and i. The distance can be represented with `long`.

Explain how you would represent this information using a jagged two-dimensional array in C efficiently. We have information about a few thousand cities to store.

Explain how you would write a function `long dist(long **d, long i, long j)` to retrieve the distance between any two cities i and j.