Okay, let's cover the essentials of arrays in C for a 1st-year undergraduate. Arrays are fundamental data structures used to store collections of elements of the same data type.

Arrays in C: Notes for a 1st Year Undergraduate

An array is a contiguous block of memory locations that store a fixed number of elements of the same data type. They provide a way to organize and access related data efficiently using a single variable name and an index.

1. Definition and Declaration

- An array is a collection of elements of the **same data type** stored in **contiguous memory locations**.
- The elements in an array are accessed using an **index** (or subscript), which typically starts from 0.
- **Declaration Syntax:**
  ```
  data_type array_name[array_size];
  ```
  - data_type: Specifies the type of elements the array will hold (e.g., int, char, float).
  - array_name: The identifier (name) you give to the array.
  - array_size: A positive integer constant expression that specifies the number of elements the array can store. This size is fixed at the time of declaration (for static arrays).
- **Examples:**
  ```
  int numbers[5];     // Declares an array named 'numbers' that can
  hold 5 integers.
  char letters[26];   // Declares an array named 'letters' that can
  hold 26 characters.
  float prices[100];  // Declares an array named 'prices' that can
  hold 100 floating-point numbers.
  ```

2. Initialization of Arrays

- You can initialize array elements when you declare the array.
  ```
  int scores[3] = {90, 85, 95}; // Initializes the 3 elements of
  'scores'.
  char vowels[5] = {'a', 'e', 'i', 'o', 'u'};
  float values[] = {1.0, 2.5, 3.7, 0.5}; // Size is automatically
  determined (4 elements).
  int zeros[5] = {0};             // Initializes the first element
  to 0, the rest to 0 by default.
  int partial[5] = {10, 20};       // Initializes the first two, the
  rest (index 2, 3, 4) to 0.
  ```
- If you don't explicitly initialize the elements of a local array, their values will be garbage (unpredictable). Global and static arrays are initialized to zero by default if no explicit

initializer is provided.

3. Accessing Array Elements

- Array elements are accessed using their index within square brackets []. The index starts from 0 and goes up to array_size - 1.
  ```
  int grades[4] = {78, 92, 88, 76};

  printf("First grade: %d\n", grades[0]);   // Output: First grade:
  78
  printf("Second grade: %d\n", grades[1]);  // Output: Second grade:
  92
  printf("Third grade: %d\n", grades[2]);   // Output: Third grade:
  88
  printf("Fourth grade: %d\n", grades[3]);  // Output: Fourth grade:
  76

  grades[1] = 95; // Modifying the value of the second element
  printf("Updated second grade: %d\n", grades[1]); // Output:
  Updated second grade: 95
  ```
- **Bounds Checking:** C **does not** perform automatic bounds checking for arrays. This means you can accidentally access memory outside the allocated range of the array (e.g., grades[4] in the example above). Doing so can lead to unpredictable behavior, program crashes, or security vulnerabilities. It's the programmer's responsibility to ensure that array indices are within the valid range (0 to array_size - 1).

4. Arrays and Loops

- Loops (especially for loops) are commonly used to iterate through the elements of an array.
  ```
  int numbers[5] = {10, 20, 30, 40, 50};
  int i;

  // Print all elements of the array
  for (i = 0; i < 5; i++) {
      printf("numbers[%d] = %d\n", i, numbers[i]);
  }

  // Calculate the sum of the elements
  int sum = 0;
  for (i = 0; i < 5; i++) {
      sum += numbers[i];
  }
  printf("Sum of elements: %d\n", sum); // Output: Sum of elements:
  150
  ```

5. Arrays and Pointers (Revisited)

- As mentioned in the pointers notes, the name of an array (without the []) acts as a pointer to the first element of the array (&array_name[0]).
```
int data[3] = {100, 200, 300};
int *ptr = data; // 'ptr' now points to data[0]

printf("Value at ptr: %d\n", *ptr);        // Output: 100
printf("Value at data[0]: %d\n", data[0]); // Output: 100

printf("Value at *(ptr + 1): %d\n", *(ptr + 1)); // Output: 200
(equivalent to data[1])
printf("Value at data[1]: %d\n", data[1]);       // Output: 200

printf("Value at ptr[2]: %d\n", ptr[2]);        // Output: 300
(pointer arithmetic with array-like access)
```
- This close relationship allows you to use pointer arithmetic to access and manipulate array elements.

6. Passing Arrays to Functions

- When you pass an array to a function, you are actually passing a pointer to the first element of the array. This means that the function can modify the original array elements.
- You typically need to pass the size of the array as another argument to the function, as the function itself doesn't know the size of the array being passed.
```
void printArray(int arr[], int size) {
    int i;
    printf("Array elements: ");
    for (i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int numbers[4] = {1, 2, 3, 4};
printArray(numbers, 4); // Passing the array and its size
```
- You can also declare the function parameter as a pointer:
```
void printArrayPtr(int *arr, int size) { // Equivalent to int
arr[]
    // ... same loop as above ...
}
```

7. Multidimensional Arrays

- C allows you to declare arrays with more than one dimension (e.g., two-dimensional arrays, which can be thought of as tables or matrices).
- **Two-Dimensional Array Declaration:**
  ```
  data_type array_name[row_size][column_size];

  int matrix[3][4]; // Declares a 2D array with 3 rows and 4
  columns.
  ```
- **Initialization of Two-Dimensional Arrays:**
  ```
  int grid[2][3] = {
      {1, 2, 3},   // First row
      {4, 5, 6}    // Second row
  };

  int table[2][2] = {1, 2, 3, 4}; // Elements are initialized in
  row-major order.
  ```
- **Accessing Elements of Two-Dimensional Arrays:** Use two indices: array_name[row_index][column_index]. Indices start from 0.
  ```
  printf("Element at row 0, column 1: %d\n", grid[0][1]); // Output:
  2
  grid[1][0] = 7; // Modifying an element
  ```
- **Iterating Through Two-Dimensional Arrays:** Nested loops are typically used.
  ```
  int matrix[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
  int i, j;

  for (i = 0; i < 3; i++) {
      for (j = 0; j < 4; j++) {
          printf("matrix[%d][%d] = %d\t", i, j, matrix[i][j]);
      }
      printf("\n");
  }
  ```

Key Takeaways about Arrays in C:

- Store a fixed number of elements of the same type.
- Elements are accessed using a zero-based index.
- Memory is allocated contiguously.
- C does not have built-in bounds checking.
- Array names often behave like pointers to the first element.
- Used extensively for organizing and manipulating collections of data.

Arrays are a fundamental building block in C programming. Understanding their properties and how to work with them is crucial for many programming tasks.