In C, a **three-dimensional array** can be used to represent a 3D structure of data. A string is simply an array of characters, so when dealing with strings in a 3D array, each "layer" (a two-dimensional array) will represent a set of strings (or a matrix of characters). You can also use **pointers** to dynamically allocate memory for such a structure, which is particularly useful when the size is not fixed at compile time.

**1. Static Three-Dimensional Array of Strings**

This is a basic 3D array where the size is fixed. Here’s an example of how to declare and work with a static 3D array of strings:

**Code Example: Static 3D Array of Strings**

#include <stdio.h>

int main() {

// Declare a 3D array of strings (3 layers, each containing 2 rows and 5 columns of characters)

char arr[3][2][5] = {

{{"Hello", "World"}, {"Good", "Morning"}},

{{"This", "is"}, {"a", "test"}},

{{"Three", "D"}, {"Arrays", "in C"}}

};

// Access and print the elements in the 3D array

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 2; j++) {

for (int k = 0; k < 5; k++) {

printf("%s ", arr[i][j][k]);

}

printf("\n");

}

printf("\n");

}

return 0;

}

**Explanation:**

* arr[3][2][5] is a 3D array with:
  + **3 layers** (first dimension).
  + **2 rows** (second dimension).
  + **5 columns** of characters (third dimension).
* Each element in the array holds a string (array of characters).

**2. Dynamic Three-Dimensional Array of Strings using Pointers**

In C, pointers can be used to create a dynamically allocated 3D array. This allows for more flexibility, especially if the size is determined at runtime.

**Code Example: Dynamic 3D Array of Strings Using Pointers**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

int layers = 3, rows = 2, columns = 5;

// Dynamically allocate memory for a 3D array of strings

char \*\*\*arr = (char \*\*\*)malloc(layers \* sizeof(char \*\*));

// Allocate memory for each layer

for (int i = 0; i < layers; i++) {

arr[i] = (char \*\*)malloc(rows \* sizeof(char \*));

// Allocate memory for each row in the layer

for (int j = 0; j < rows; j++) {

arr[i][j] = (char \*)malloc(columns \* sizeof(char \*));

}

}

// Assign strings to the 3D array

arr[0][0][0] = "Hello";

arr[0][0][1] = "World";

arr[0][1][0] = "Good";

arr[0][1][1] = "Morning";

arr[1][0][0] = "This";

arr[1][0][1] = "is";

arr[1][1][0] = "a";

arr[1][1][1] = "test";

arr[2][0][0] = "Three";

arr[2][0][1] = "D";

arr[2][1][0] = "Arrays";

arr[2][1][1] = "in C";

// Print the 3D array

for (int i = 0; i < layers; i++) {

for (int j = 0; j < rows; j++) {

for (int k = 0; k < columns; k++) {

printf("%s ", arr[i][j][k]);

}

printf("\n");

}

printf("\n");

}

// Free dynamically allocated memory

for (int i = 0; i < layers; i++) {

for (int j = 0; j < rows; j++) {

free(arr[i][j]);

}

free(arr[i]);

}

free(arr);

return 0;

}

**Explanation:**

1. **Dynamic Memory Allocation**:
   * The malloc function is used to allocate memory for the 3D array.
   * The memory allocation for each level is done in a nested manner, starting with allocating memory for layers, then for rows within each layer, and finally for columns within each row.
2. **Assigning Strings**:
   * Here, instead of assigning character arrays directly to the 3D array (like in a static array), we assign string literals (which are pointers) to the elements of the 3D array.
3. **Freeing Memory**:
   * The memory allocated dynamically is freed using free() to avoid memory leaks. Each level of the array (columns, rows, and layers) must be freed in reverse order of allocation.

**Key Concepts in the Code:**

1. **Pointer Arithmetic**: By using pointers, the code can dynamically allocate memory for each part of the 3D array.
2. **String Literals**: Strings in C are pointers to arrays of characters, so we store string literals (such as "Hello") directly in the array.
3. **Memory Management**: Memory is manually allocated and deallocated using malloc and free functions.

**3. Accessing 3D Array Elements with Pointers**

When using pointers, you can access elements in a 3D array as follows:

**printf("%s\n", arr[layer][row][column]);**

This will print the string stored at the specified layer, row, and column.

**Summary**

* **Static 3D Array of Strings**: Declares a fixed-size 3D array and stores string literals directly.
* **Dynamic 3D Array of Strings**: Uses pointers to dynamically allocate memory for a 3D array, allowing for flexible memory management.
* **Pointers and Strings**: Strings in C are essentially pointers to character arrays, and pointer-based manipulation offers greater flexibility, especially when dealing with dynamic data.

**HASHING**

Hashing is a technique used to uniquely identify a data item by mapping it to a fixed-size index (hash value) in an array. This process helps in performing efficient data lookups, insertions, and deletions. A **hash function** takes an input (or "key") and returns an index where the corresponding value can be stored or retrieved.

**Hash Function**: A function that maps data to a fixed-size array index.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define SIZE 10 // Define the size of the hash table

// Define the structure for a linked list node

struct Node {

int key;

struct Node\* next;

};

// Define the structure for the hash table

struct HashTable {

struct Node\* table[SIZE]; // Array of linked lists (for chaining)

};

// Hash function to compute index for a given key

int hashFunction(int key) {

return key % SIZE;

}

// Initialize the hash table

void initHashTable(struct HashTable\* ht) {

for (int i = 0; i < SIZE; i++) {

ht->table[i] = NULL; // Initialize all slots to NULL

}

}

// Insert a key into the hash table

void insert(struct HashTable\* ht, int key) {

int index = hashFunction(key);

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->key = key;

newNode->next = ht->table[index]; // Insert at the head of the list

ht->table[index] = newNode; // Update the table to point to the new node

}

// Search for a key in the hash table

struct Node\* search(struct HashTable\* ht, int key) {

int index = hashFunction(key);

struct Node\* current = ht->table[index];

while (current != NULL) {

if (current->key == key) {

return current; // Key found

}

current = current->next; // Move to the next node in the list

}

return NULL; // Key not found

}

// Delete a key from the hash table

void delete(struct HashTable\* ht, int key) {

int index = hashFunction(key);

struct Node\* current = ht->table[index];

struct Node\* prev = NULL;

while (current != NULL) {

if (current->key == key) {

if (prev == NULL) {

ht->table[index] = current->next; // Remove the first node

} else {

prev->next = current->next; // Bypass the node to delete it

}

free(current); // Free memory of the deleted node

return;

}

prev = current;

current = current->next;

}

printf("Key %d not found\n", key); // If key is not found

}

// Print the hash table

void printHashTable(struct HashTable\* ht) {

for (int i = 0; i < SIZE; i++) {

printf("Index %d: ", i);

struct Node\* current = ht->table[i];

while (current != NULL) {

printf("%d -> ", current->key);

current = current->next;

}

printf("NULL\n");

}

}

int main() {

struct HashTable ht;

initHashTable(&ht); // Initialize the hash table

// Insert some keys into the hash table

insert(&ht, 10);

insert(&ht, 20);

insert(&ht, 30);

insert(&ht, 25);

insert(&ht, 35);

// Print the hash table

printf("Hash Table:\n");

printHashTable(&ht);

// Search for a key in the hash table

int keyToSearch = 25;

struct Node\* result = search(&ht, keyToSearch);

if (result != NULL) {

printf("Found key %d in the hash table\n", keyToSearch);

} else {

printf("Key %d not found in the hash table\n", keyToSearch);

}

// Delete a key from the hash table

int keyToDelete = 30;

delete(&ht, keyToDelete);

printf("\nHash Table after deleting key %d:\n", keyToDelete);

printHashTable(&ht);

return 0;

}

Output:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define SIZE 10 // Define the size of the hash table

// Define the structure for a linked list node

struct Node {

int key;

struct Node\* next;

};

// Define the structure for the hash table

struct HashTable {

struct Node\* table[SIZE]; // Array of linked lists (for chaining)

};

// Hash function to compute index for a given key

int hashFunction(int key) {

return key % SIZE;

}

// Initialize the hash table

void initHashTable(struct HashTable\* ht) {

for (int i = 0; i < SIZE; i++) {

ht->table[i] = NULL; // Initialize all slots to NULL

}

}

// Insert a key into the hash table

void insert(struct HashTable\* ht, int key) {

int index = hashFunction(key);

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->key = key;

newNode->next = ht->table[index]; // Insert at the head of the list

ht->table[index] = newNode; // Update the table to point to the new node

}

// Search for a key in the hash table

struct Node\* search(struct HashTable\* ht, int key) {

int index = hashFunction(key);

struct Node\* current = ht->table[index];

while (current != NULL) {

if (current->key == key) {

return current; // Key found

}

current = current->next; // Move to the next node in the list

}

return NULL; // Key not found

}

// Delete a key from the hash table

void delete(struct HashTable\* ht, int key) {

int index = hashFunction(key);

struct Node\* current = ht->table[index];

struct Node\* prev = NULL;

while (current != NULL) {

if (current->key == key) {

if (prev == NULL) {

ht->table[index] = current->next; // Remove the first node

} else {

prev->next = current->next; // Bypass the node to delete it

}

free(current); // Free memory of the deleted node

return;

}

prev = current;

current = current->next;

}

printf("Key %d not found\n", key); // If key is not found

}

// Print the hash table

void printHashTable(struct HashTable\* ht) {

for (int i = 0; i < SIZE; i++) {

printf("Index %d: ", i);

struct Node\* current = ht->table[i];

while (current != NULL) {

printf("%d -> ", current->key);

current = current->next;

}

printf("NULL\n");

}

}

int main() {

struct HashTable ht;

initHashTable(&ht); // Initialize the hash table

// Insert some keys into the hash table

insert(&ht, 10);

insert(&ht, 20);

insert(&ht, 30);

insert(&ht, 25);

insert(&ht, 35);

// Print the hash table

printf("Hash Table:\n");

printHashTable(&ht);

// Search for a key in the hash table

int keyToSearch = 25;

struct Node\* result = search(&ht, keyToSearch);

if (result != NULL) {

printf("Found key %d in the hash table\n", keyToSearch);

} else {

printf("Key %d not found in the hash table\n", keyToSearch);

}

// Delete a key from the hash table

int keyToDelete = 30;

delete(&ht, keyToDelete);

printf("\nHash Table after deleting key %d:\n", keyToDelete);

printHashTable(&ht);

return 0;

}